INFORMATIK HANDBUCH FÜR DIE ABSCHLUSSPRÜFUNG

DEUTSCHSPRACHIGER STUDIENGANG DER INFORMATIK

Allgemeine Thematik

Teil 1. Algorithmen und Programmierung

1. Suchen (linear und binär), Sortieren (Selection, Bubblesort, Quicksort). Backtracking.
2. OOP in Programmiersprachen (Python, C++, Java, C#): Klassen und Objekte. Membership in Klassen und Zugang. Konstruktor und Destruktor.
3. Relationen zwischen Klassen. Abgeleitete Klassen und Vererbung. Polymorphismus. Abstrakte Klassen.
4. Klassendiagramme und UML Interaktionen zwischen Objekten: Pakete, Klassen und Interfaces. Beziehungen zwischen Klassen und Interfaces. Objekte. Messages
5. Listen und assoziative Datenfelder
6. Aufgaben

Teil 2: Datenbanken

1. Relationale Datenbanken. Die ersten drei Normalformen einer Relation.
2. Datenbanken Abfragen mit Hilfe der relationalen Algebra
3. Datenbanken Abfragen in SQL (Select)

Teil 3: Betriebssysteme

1. Struktur und Funktionen von Betriebssystemen
2. Dateisysteme : hierarchische Struktur, Links Fallstudien Unix (Linux)
3. Unix Prozesse: Prozesserstellung, Systemaufrufe fork, exec, exit, wait; Kommunikation zwischen Prozessen über pipe und FIFO
4. Interpreter von Batch-Dateien mit sh Illustration (Unix)

1.1. Suchen und Sortieren

1.1.1 Suchen

Daten werden in einer Reihenfolge gespeichert. Das Ziel der Suche ist es, ein bestimmtes Element zu finden, von dem der zugehörige Suchschlüssel bekannt ist. Wenn die Suche erfolgreich ist, wird als Rückgabewert der Index des Elements der Folge zurückgegeben.

Seien $k_1, k_2, .., k_n$ die Schlüssel der Elemente und a der Suchschlüssel. Das Problem reduziert sich auf die Suche des Indexes p so dass $a = k_p$ gilt. Es ist allgemein üblich, dass die Schlüssel sortiert sind: $k_1 < k_2 < ... < k_n$.

Manchmal, wenn die Schlüssel sortiert sind aber die Suche nicht erfolgreich ist, wollen wir der Möglicheeinfügungsindex eines Elements wissen, so dass die Reihenfolge durch die Einfügung unverändert bleibt.

Die vollständige Spezifikation der Suche ist:

Daten a,n,(k_i, i=1,n);
Vorbedingung: n∈N, n≥1, und $k_1 < k_2 < .... < k_n$;
Ergebnis p;
Nachbedingung: (p=1 und a ≤k_1) oder (p=n+1 und a > k_n) oder (1<p≤n) und (k_{p-1} < a ≤k_p).

1.1.1.1 Lineare Suche

Das erste Verfahren ist eine sequentielle Überprüfung aller Elemente des Feldes. Wir betrachten drei Fälle: $a \leq k_1$, $a > k_n$, $k_1 < a \leq k_n$ und die effektive Suche.

```
Subalgorithm SearchSeq (a,n,K,p) is:
                                    {n∈N, n≥1 and k1 < k2 < .... < kn}
                                {Search p such that: (p=1 and a≤k1) or}
                        { (p=n+1 and a>kn) or (1<p≤n) and (kp-1<a≤kp)}
                                                {Case "not yet found"}

      Let p:=0;
      If a≤k1 then p:=1 else
            If a>kn then p:=n+1 else
                  For i:=2; n do
                        If (p=0) and (a≤ki) then p:=i
                        endif
                  endfor
            endif
      endif
sfsub
```

Wir beobachten, dass diese Methode zu n-1 Vergleichsschritte im schlechtesten Fall führt, weil i Werte von 2 bis n nimmt. Die n Schlüssel teilen die reelle Achse in n+1 Intervalle. Wenn a zwischen k_1 und k_n liegt, ist die Anzahl der Vergleiche immer noch n-1. Wenn a außerhalb $[k_1,k_n]$ liegt, gibt es am höchstens 2 Vergleichsschritte. Somit hat die mittlere Komplexität eine gleiche Größenordnung wie die schlechteste Komplexität.

Es gibt viele Situationen wann der Algorithmus nutzlose Vergleichsschritte macht. Wenn der Schlüssel schon identifiziert ist, ist die Fortführung der For-Schleife nutzlos. In anderen Worten: es ist erwünscht, dass die **For**-Schleife mit einer **While**-Schleife ersetzt ist. Der Algorithmus ist:

```
Subalgorithm SearchSucc (a,n,K,p) is:
                                        {n∈N, n≥1 and k1 < k2 < .... < kn}
                                  {Search p such that: (p=1 and a≤k1) or}
                            { (p=n+1 and a>kn) or (1<p≤n) and (kp-1<a≤kp)}

      Let p:=1;
      If a>k1 then
            while p≤n und a>kp do
                  p:=p+1
            endwh
      endif
sfsub
```

Der Algorithmus *SearchSucc* macht im schlechtesten Fall n Vergleichsschritte. Aber die Anzahl der Vergleiche ist durchschnittlich auf die Hälfte reduziert und damit ist die mittlere Laufzeit *SearchSucc* die Gleiche wie *SearchSeq*.

1.1.1.2 Binäre Suche

Die binäre Suche ist mehr effizienter als die zwei zuvor beschriebenen Methoden, weil es eine einfache Form des Schemas "Teilen und Herrschen" benutzt. Zuerst wird das mittlere Element des Feldes überprüft. Es kann kleiner, größer oder gleich dem gesuchten Element sein. Ist es kleiner als das gesuchte Element, muss das gesuchte Element in der hinteren Hälfte stecken, falls es sich dort überhaupt befindet. Ist es hingegen größer, muss nur in der vorderen Hälfte weiter gesucht werden. Die jeweils andere Hälfte muss nicht mehr betrachtet werden. Wenn Ist es gleich mit dem gesuchten Element, wird die Suche beendet. Die Länge des Suchbereiches wird so von Schritt zu Schritt halbiert. Ein rekursives Verfahren ist:

```
Subalgorithm SearchBin (a,n,K,p) is:
                                        {n∈N, n≥1 and k1 < k2 < .... < kn}
                                  {Search p such that: (p=1 and a≤k1) or}
```

```
                        { (p=n+1 and a>k_n) or (1<p≤n) and (k_p-1<a≤k_p)}

      If a≤K_1
            then p:=1
      else
            If a>K_n
                  then p:=n+1
                  else p:=BinarySearch(a,K,1,n)
            endif
      endif
sfsub

Function BinarySearch (a, K, Left, Right) is:
      If Left≥Right-1
            then BinarySearch:=Rigt
            else m:=(Left+Right) Div 2;
                  If a≤K_m
                        then BinarySearch:=BinarySearch(a,K,Left,m)
                        else BinarySearch:=BinarySearch(a,K,m,Right)
                  endif
      endif
sffunc
```

Die *Left* und *Right* Variablen beschreiben in *BinarySearch* die Enden des Intervalls und *m* beschreibt den Mittelpunkt. Um in einem Feld mit *n* Einträgen die An- oder Abwesenheit eines Schlüssels festzustellen, werden maximal $log_2 n$ Vergleichsschritte benötigt. Somit ist die schlechteste Komplexität proportional mit $log_2 n$.

Ein iteratives Verfahren ist:

```
Function BinarySearchN (a,K,Left,Right) is:
      While Right-Left>1 do
            m:=(Left+Right) Div 2;
            If a≤K_m then
                  Right:=m
                  else Left:=m
            endif
      endwh
      BinarySearchN:=Right
endfunc
```

1.1.2 Sortieren

Ein Sortierverfahren ist ein Algorithmus, der die Elemente eines Feldes ordnet, damit die Schlüssel in einer aufsteigende (absteigende) Reihenfolge sortiert werden. Die Spezifikation des Sortierens ist:

Daten n,K; *{K=(k_1,k_2,...,k_n)}*
Vorbedingung: $k_i \in R$, i=1,n;
Ergebnis K';
Nachbedingung: K' ist eine Permutation von K
$k'_1 \leq k'_2 \leq ... \leq k'_n$.

1.1.2.1 Selectionsort

Selectionsort sucht das kleinste Element in das Feld und vertauscht es mit dem aktuellen Element. Danach ist das Array bis zu dieser Position sortiert. Anschließend wird das Verfahren so lange wiederholt, bis das gesamte Array abgearbeitet worden ist.

```
Subalgorithm SelectionSort(n,K) is:
                                              {Do a permutation of the}
                                        {n components of vector K such}
                                                {that k1≤k2≤...≤kn }
      For i:=1; n-1 do
            Let ind:=i;
            For j:=i+1; n do
                 If kj < kind then ind:=j endif
            endfor
            If i<ind then t:=ki; ki:=kind; kind:=t endif
      endfor

sfsub
```

Wir beobachten, dass die Anzahl der Vergleiche (n-1)+(n-2) +...+2+1=n(n-1)/2 unabhängig von den Daten ist. Somit liegt der Selectionsort in der Komplexitätsklasse $O(n^2)$.

1.1.2.2 Bubblesort

Bubblesort vertauscht zwei aufeinanderfolgende Elemente, wenn ihre Reihenfolge nicht stimmt. Das Verfahren durchläuft die Menge gegebenenfalls mehrmals, bis bei einem Durchlauf keine Vertauschungen mehr durchgeführt werden mussten.

```
Subalgorithm BubbleSort (n,K) is:
      Repeat
            Let kod:=0;                        {Hypothesis "is sorted"}
            For i:=2; n do
                  If ki-1 > ki then
                        t := ki-1;
```

```
                    k_{i-1} := k_i;
                    k_i:=t;
                    kod:=1                          {Not sorted yet!}
               endif
          endfor
     until kod=0 endrep                                        {Sorted}
sfsub
```

Die Anzahl der Vergleiche ist im schlechtesten Fall (n-1)+(n-2) +...+2+1=n(n-1)/2. Somit liegt der Bubblesort in der Komplexitätsklasse $O(n^2)$. Ein optimiertes Verfahren ist:

```
Subalgorithm BubbleSort (n,K) is:
     Let s:=0;
     Repeat
          Let kod:=0;                       {Hypothesis "is sorted"}
          For i:=2; n-s do
               If k_{i-1} > k_i then
                    t := k_i-1;
                    k_{i-1} := k_i;
                    k_i:=t;
                    kod:=1                          {Not sorted yet!}
               endif
          endfor
          Let s:=s+1;
     until kod=0 endrep                                   {Sorted}
sfsub
```

1.1.2.3 Quicksort

Quicksort ist ein schneller, rekursiver Sortieralgorithmus, der eine Form des Schemas "Teilen und Herrschen" benutzt. Um den Feld zu sortieren, die Methode *QuickSort* ruft die Methode `QuickSortRec(K,1,n)` auf. *K* is der Feld und *n* ist die Anzahl der Elemente in *K*.

```
Subalgorithm QuickSort (n,K) is:
     Call QuickSortRec(K,1,n)
sfsub
```

Die Methode `QuickSortRec(K,left,right)` sortiert die Reihenfolge $k_{left},k_{left+1},...,k_{right}$. Vor dem tatsächlichen Sortieren, die Reihenfolge aufgeteilt ist, so dass das k_{left} Element (das Pivot) die letzte Position des Feldes besitzt. Falls i diese Position ist, wird die Reihenfolge geordnet so dass:

$k_j \leq k_i \leq k_l$, für $Left \leq j < i < l \leq Right$ (*)

Sobald die Aufteilung erreicht ist, müssen wir zwei rekursiven Aufrufe `QuickSortRec(K, left, i-1)`, `QuickSortRect(K,i+1,Right)` machen, um die beiden Teilen zu

Sortieren. Natürlich nur wenn die Reihenfolgen mehr als zwei Elemente haben. Ansonsten eine Reihenfolge mit einem Element ist sortiert.

Der Algorithmus ist:

```
Subalgorithm QuickSort (K, Left, Right) is:
      Let i := Left; j := Right; a := k_i;
      Repeat
            While k_j≥a and (i < j) do j := j - 1 endwh
            k_i := k_j;
            While k_i≤a and (i < j) do i := i + 1 endwh k_j := k_i ;
      until i = j endrep
      Let ki := a;
      If St < i-1 then Call QuickSort(K, St, i - 1) endif
      If i+1 < Dr then Call QuickSort(K, i + 1, Dr) endif
endsub
```

Im Durchschnitt führt der Quicksort $O(n\log_2 n)$ Vergleiche durch. Im schlechtesten Fall werden $O(n^2)$ Vergleiche durchgeführt, was aber in der Praxis sehr selten vorkommt.

1.1.3 Backtracking

Backtracking ist geeignet für Probleme mit vielen Lösungen. Es hat jedoch den Nachteil, dass es eine exponentielle Laufzeit hat. Erstmal diskutieren wir zwei Beispiele und danach geben wir allgemeine Verfahren für die Methode ab.

Aufgabe 1. Gegeben Sei *n*, eine natürliche Zahl. Generieren Sie alle Permutationen von *1, 2,..., n*.

Eine Lösung für n=3 ist

```
Subalgorithm Permutations1 is:
      For i1 := 1;3 execute
            For i2 := 1;3 execute
                  For i3 := 1;3 execute
                        Let possible := (i1, i2, i3)
                        If components of the possible array are
                        distinct
                              then Print possible
                        endif
                  endfor
            endfor
      endfor
endsub
```

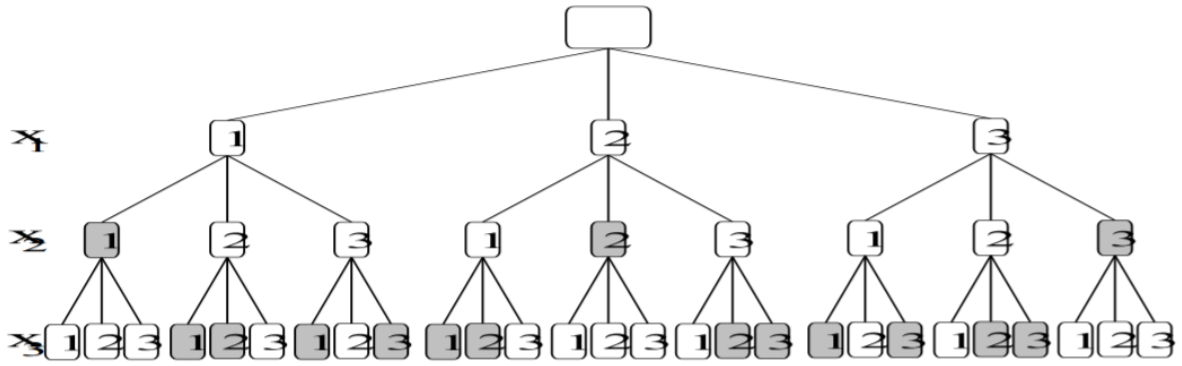


Abbildung 1.1 Graphische Beschreibung des Kartesischen Produkts $\{1,2,3\}^3$

Anmerkungen zum Algorithmus *Permutations1*:

- für den allgemeinen Fall sollte man *n* verschaltete **For**-Schleifen schreiben
- die gesamte Anzahl des überprüften Feldes ist 3^3 und im Allgemeinen n^n. Abbildung 1.1 zeigt die *möglichen* überprüften Felder
- der Algorithmus generiert die Komponenten des Feldes und danach überprüft ob es eine Permutation ist

Eine Möglichkeit den Algorithmus zu verbessern ist, die Überprüfung während der Erzeugung des Feldes zu machen. Somit sind Felder, die nicht zu eine Lösung führen nicht aufgestellt. Zum Beispiel, wenn die erste Komponente des Feldes 1 ist, ist es nutzlos, um der zweiten Komponente 1 zuzuweisen. Auf diese Weise können wir die Erzeugung des Feldes der Typ (1,...) für großen Zahlen begrenzen. Zum Beispiel (1,3,...) ist ein *mögliches Feld* (zu einer Lösung führen kann), aber (1,1,...) ist nicht. Das Feld (1, 3, ...) erfüllt bestimmte *fortsetzung Bedingungen*: es hat distinkten Komponenten. Die grauen Knoten in Abbildung 1.1 zeigt Werte, die zu einer Lösung nicht führen kann.

Wir werden ein allgemeines Verfahren für Backtracking beschreiben und danach sehen wir wie es für die Probleme in diesem Teil benutzen werden kann. Anmerkungen zum Backtracking:

1. der Lösungsraum: $S = S_1 \times S_2 \times ... \times S_n$;
2. *possible* ist ein Feld, das eine Lösung beschreibt;
3. *possible[1..k]*$\in S_1 \times S_2 \times ... \times S_n$ ist das Sub-Feld von Kandidaten-Lösungen. Es könnte oder nicht zu eine Lösung führen. *k* ist die Anzahl der schon aufgestellten Elemente der Lösungen;
4. *possible[1..k]* verspricht, dass zu einer Lösung führt wenn die Bedingungen erfüllt sind (Abbildung 1.2);
5. *solution (n, k, possible)* ist eine Funktion die prüft, ob ein Kandidat-Feld *possible[1..k]* ,eine Lösung wäre.

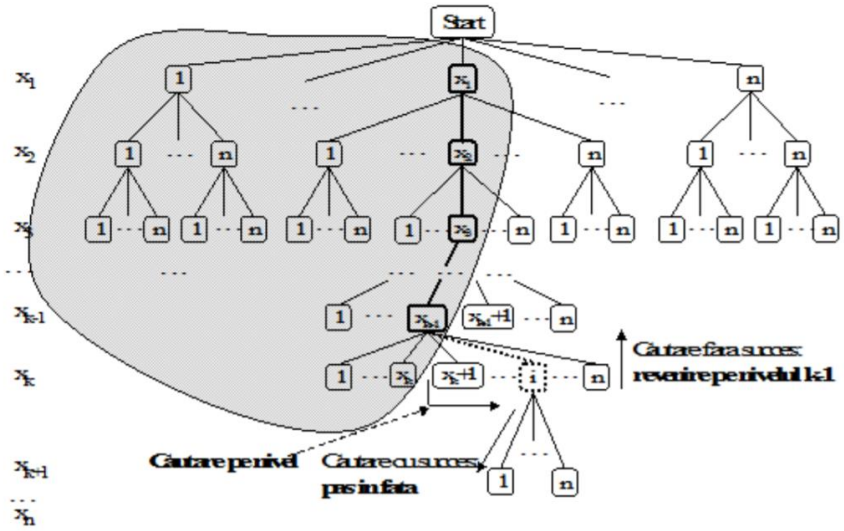


Abbildung 1.2 Der Suchraum des Permutation-Problems

Das Verfahren wird im Folgenden beschrieben:

```
Subalgorithm Backtracking(n) is:                    {draft version }
      Let k = 1;
      @Initialise the search for index k (= 1)
      While k > 0 do
            {possible[1..k-1] is a solution candidate }
            @Sequentially search for index k a value v to extend the
            subarray possible[1..k-1] such that possible[1..k] is
            still a solution candidate
            If the search is successful
                  then Let possible[k] := v;
                        {possible[1..k] is a solution candidate}
                        If solution (n, k, possible)
                              {found a solution; we are still on level
                              k}
                              then Print possible[1..k]
                                    else @Initialize the search for
                              index k+1 {a potential array }
                                    Let k = k + 1
                                    {step forward on level k+1}
                              endif
                        else k = k - 1
                        {step backward (backtrack to level k-1)}
            endif
      endwh
endSub
```

Um das Verfahren zu Ende zubringen, müssen die Non-Standard Teile einfügt werden. Die Boolean-Funktion,

```
      condToContinue (k, possible, v)
```

die prüft, ob ein Sub-Feld *possible[1..k]* mit *v* erweitert, eine Lösung wäre. Die Funktion,

```
init (j)
```

die ein fiktives Element von S_j zurückgibt. Dies Element zeigt, dass kein Element von S_j auf Suchebene *j* schon ausgewählt ist. Die Boolean-Funktion

```
next (j, v, new)
```

Falls einen neuen Wert von S_j ausgewählt werden kann, gibt die Funktion *true* zurück. Falls das Element nicht existiert, gibt die Funktion *false* zurück. Der aktualisierte Algorithmus ist:

```
Subalgorithm Backtracking(n) is:                    {final version }
      Let k = 1;
      possible[1] := init(1);
      While k > 0 do
            Let found := false; v := possible[k] ;
            While next (k, v, new) and not found do
                  Let v := new;
                  If condToContinue (k, possible, v)
                        then found := true
                  endif
            endwh
            If found then Let possible[k] := v
            {possible[1..k] is potential}
                  if solution (n, k, possible)
                  {found a solution; we are still on level k}
                        then Print possible[1..k]
                        else Let k = k + 1;        {a potential array}
                              possible[k] := init(k);
                              {step forward on level k+1}
                  endif
                        else k = k - 1
                        {step backward (backtrack to level k-1)}
            endif
      endwh
endSub
```

Der Suchansatz eines Wertes und die Funktionen *solution* und *condToCont* hängen vom Problem ab. Zum Beispiel diese Funktionen sind für das Permutation-Problem:

```
function init(k) is:
      init:= 0
endfunc

function next (k, v, new) is:
      if v < n
```

```
            then next := true;
                 new:= v + 1
            else next := false
      endif
endfunc

function condToContinue (k, possible, v) is:
      kod:=True; i:=1;
      while kod and (i<k) do
            if possible[i] = v then kod := false endif
      endwh
      condToContinue := kod
endfunc

function solution (n, k, possible) is:
      solution := (k = n)
endfunc
```

Eine rekursive Implementierung des Backtracking mit den gleichen Hilfsfunktionen ist:

```
Subalgorithm backtracking(n, k, possible) is:
{possible[1..k] is potential}
      if solution(n, k, possible)
      {a solution; terminate the recursive call}
            then print possible[1..k]
            {else, stay on same level}
            else for each value v possible for possible[k+1] do
                  if condToContinue(k+1, possible, v)
                        then possible[k+1] := v
                             backtracking (n, k+1, possible)
                             {step forward}
                  endif
            endfor
      endif
      {terminate backtracking(n, k, possible) call}
endsub                                          {so, step backward}
```

Das problem ist durch einen Aufruf *backtracking(n, 0, possible)* gelöst.

1.2. OOP in Programmiersprachen

1.2.1 Klassen

1.2.1.1 Datenschutz in der modularen Programmierung

Die Entwicklung der Programme bedeutet in der prozeduralen Programmierung die Nutzung der Funktionen und Prozeduren. In der C Programmiersprache gibt es Funktionen mit oder

ohne einen Rückgabewert, statt Funktionen und Prozeduren. Aber für große Anwendungen ist ein gewisses Maß dem Datenschutz erwünscht. Das bedeutet, dass nur wenige Funktionen zu den Daten Zugang haben. Datenschutz kann in der modularen Programmierung mit statische Zuordnung erreichen. Wenn eine Variable außerhalb einer Funktion deklariert wird, ist die Variable in der ganzen Datei sichtbar. Aber existiert es nicht mehr außerhalb der Datei.

In dem nächsten Beispiel wollen wir über eine Vektorrechnungsanwendung betrachten. Schreiben Sie ein Modul, das die folgenden Funktionen enthält: Initialisierung, Freigabe, Quadrieren und Print. Eine mögliche Implementierung ist:

```
#include <iostream>
using namespace std;

static int* e; //vector elements static
int d; //vector size

void init(int* e1, int d1) //initialization {
      d = d1;
      e = new int[d];
      for(int i = 0; i < d; i++)
            e[i] = e1[i];
}

void destroy() //disposing occupied memory {
      delete [] e;
}

void squared() // raising to the power two {
      for(int i = 0; i < d; i++)
            e[i] *= e[i];
}

void print() //printing {
      for(int i = 0; i < d; i++)
            cout << e[i] << ' ';
            cout << endl;
}
```

Das Modul soll einzeln kompiliert werden und eine Objektdatei ist erstellt.

```
extern void init( int*, int); //extern may be omitted
extern void destroy();
extern void squared();
extern void print();
//extern int* e;

int main() {
      int x[5] = {1, 2, 3, 4, 5};
```

```
	init(x, 5);
	squared();
	print();
	destroy();
	int y[] = {1, 2, 3, 4, 5, 6};
	init(y, 6);
	//e[1]=10; error, data are protected
	squared();
	print();
	destroy();
	return 0;
}
```

Obwohl das Main-Program zwei Vektoren benutzt, kann man nicht die gleichzeitig benutzen. Damit kann das Modul nicht erweitert werden, um Vektoraddition zu rechnen. Abstrakte Datentypen werden eingeführt, um diesen Nachteil zu beseitigen.

1.2.1.2 Abstrakte Datentypen

Abstrakte Datentypen erlauben eine enge Beziehung zwischen Daten des Problems und deren Operationen. Eine Deklaration einem abstrakten Datentyp ist gleich als eine Struktdeklaration, die Daten und aller zulässigen Operationen zusammenbringt. Ein abstrakter Datentyp für die Vektorrechnungsanwendung:

```
struct vect {
	int* e;
	int d;

	void init(int* e1, int d1);
	void destroy() {
		delete [] e;
	}

	void squared();
	void print();
};
```

Funktionen sind in einer Spezifikation Methoden genannt, während Daten Attribute genannt sind. Wenn die Implementierung einer Methode in der Spezifikation liegt (wie die destroy Methode), ist die Methode inline genannt. Wenn eine Methode außerhalb der Spezifikation implementiert wird, muss man der Scope Resolution Operator benutzen. Zum Beispiel die init, squared und print Methoden sind wie folgt implementiert:

```
void vect::init(int *e1, int d1) {
	d = d1;
	e = new int[d];
```

```
      for(int i = 0; i < d; i++)
            e[i] = e1[i];
}

void vect::squared() {
      for(int i = 0; i < d; i++)
            e[i] *= e[i];
}

void vect::print() {
      for(int i = 0; i < d; i++)
            cout << e[i] << ' ';
      cout << endl;
}
```

Obwohl der oben beschriebene Ansatz eine enge Beziehung zwischen Daten (des Problems) und deren Operationen erreicht hat, erfolgt der Zugriff nicht nur über die festgelegten Operationen (die Daten sind nicht geschützt oder nach außen gekapselt). Um diesen Nachteil zu beseitigen, kann man Klassen benutzen.

1.2.1.3 Deklaration von Klassen

Eine Klasse ist ein benutzerdefinierter Datentyp, der ähnlich einer Structure in C aus Methoden und Daten aufgebaut ist. Für den Zugriff zu den einzelnen Komponenten (Methoden, Daten) können Beschränkungen festgelegt werden. Zur Kennzeichnung der Zugriffsbeschränkungen dienen Zugriffs-Specifier: *public, private, protected*. Zu den *private* und *protected* Komponenten ist ein Zugriff von außen nicht zulässig (die Daten sind geschützt). Zu den *public* Komponenten kann ohne Beschränkung, also auch von außen, zugegriffen werden. Defaultmäßig, ohne Angabe eines Zugriffs-Specifier, sind die Komponenten einer Klasse *private*. Einziger Unterschied zwischen *Class* und *Struct*: Defaultmäßig sind die Komponenten mittels struct definierten Klassen *public*.

Eine mögliche Vektor-Klasse kann wie folgt erstellt sein:

```
class vector {
      int* e; //vector elements
      int d; //vector size

public:
      vector(int* e1, int d1);
      ~vector() {
            delete [] e;
      }

      void squared();
```

```
      void print();
};
```

Daten (*e* und *d*) sind *private* definiert, während Funktionen zur Manipulation dieser Daten public definiert sind. Zu der private Komponente dürfen nur Funktionen, die selbst Komponente derselben Klasse sind, sowie *Freund-Funktionen* dieser Klasse zugreifen.

Ein durch eine Klassendefinition eingeführter Klassenname kann zur Definition von Instanzen dieser Klasse (Variablen) verwendet werden. Diese Instanzen bezeichnet man üblicherweise als Objekte:

```
      class_name list_of_objects;
```

Zum Beispiel, ein Vektorinstanz:

```
      vector v;
```

Bei der Erzeugung eines Objekts der Klasse wird eine spezielle Methode (*der Konstruktor*) automatisch aufgerufen. Ein Destruktor ist das Gegenstück zum Konstruktor. Er wird automatisch bei Beendigung der Lebensdauer eines Objekts der Klasse aufgerufen, unmittelbar vor der Freigabe des Speicherplatzes für das Objekt. Zum Beispiel:

```
      vector(int* e1, int d1);
```

ist ein Konstruktor und

```
      ~vector() { delete [] e; }
```

ist ein Destruktor.

In das oben beschriebene Beispiel, ist der Destruktor innerhalb der Klassendefinition definiert aber der Konstruktor ist nicht. Der Destruktor wird dann vom Compiler in eine Inline-Funktion übergesetzt. Um eine Methode außerhalb der Klassendefinition zu implementieren, muss man der Scope Resolution Operator benutzen.

1.2.1.4 Member-Funktionen. Der Pointer this

Memberfunktionen können nur über ein Objekt aufgerufen werden. Dies erfolgt mit dem Objektname und des Operators “.“ bzw. eines Objektpointers und des Operators “-> “. Zum Beispiel:

```
vector v;
vector* p;

vector v;
vector* p;
```

Innerhalb der Methoden kann man auf alle Komponenten des aktuellen Objekts allein mit dem Komponentennamen zugegriffen werden. Dies ist möglich, weil der Compiler beim Funktionsaufruf ein spezieller Pointer automatisch erzeugt, der auf das aktuelle Objekt verbunden wird. Dieser Pointer steht unter dem Namen *this* zur Verfügung. Er ist ein konstanter Pointer, der beim Funktionsaufruf die Attribute und Methoden des aktuellen Objektes bestimmen kann. Zum Beispiel, innerhalb der print Methode wird ein Attribut *d* als *this->d* übersetzt.

Der Pointer *this* kann direkt zugegriffen werden.

1.2.1.5 Der Konstruktor

Die Erzeugung eines Objekts der Klasse ist durch eine Aufruf einer speziellen Methode (der Konstruktor) gemacht. Der Konstruktor hat gleichen Name wie der Klassenname. Konstruktoren lassen sich analog zu normalen Methoden auch überladen. Natürlich soll die Anzahl oder der Typ von Parametern verschieden sein, um der Kompiler den richtig Konstruktor zu wählen.

Konstruktoren sind normalerweise public.

Konstruktoren haben kein Funktionstyp (auch nicht void), damit sie auch keine Rückgabe eines Funktionswertes zurückgeben können

Im Folgenden beschreiben wir eine Klasse mit zwei Attribute (nachname, vorname) und eine Print-Methode.

File `person.h`:

```
class person {
      char* lname;
      char* fname;
public:
      person(); //default constructor
      person(char* ln, char* fn); //constructor
      person(const person& p1); //copy constructor
      ~person(); //destructor
      void print();
};
```

File `person.cpp`:

```
#include <iostream>
#include <cstring>
#include "person.h"

using namespace std;
```

```
person::person() {
      lname = new char[1];
      *lname = 0;
      fname = new char[1];
      *fname = 0;
      cout << "Calling default constructor." << endl;
}

person::person(char* ln, char* fn) {
      lname = new char[strlen(ln)+1];
      fname = new char[strlen(fn)+1];
      strcpy(lname, ln); strcpy(fname, fn);
      cout << "Calling constructor (lname, fname).\n";
}

person::person(const person& p1) {
      lname = new char[strlen(p1.lname)+1];
      strcpy(lname, p1.name);
      fname = new char[strlen(p1.fname)+1];
      strcpy(fname, p1.fname);
      cout << "Calling copy constructor." << endl;
}
person::~person() {
      delete[] lname;
      delete[] fname;
}

void person::print() {
      cout << fname << ' ' << lname << endl;
}
```

File `personTest.cpp`:

```
#include "person.h"
int main() {
      person A; //calling default constructor
      A.print();
      person B("Stroustrup", "Bjarne");
      B.print();
      person *C = new person("Kernighan","Brian");
      C->print();
      delete C;
      person D(B); //equivalent to person D = B;
                   //calling copy constructor
      D.print();
      return 0;
}
```

Wir beobachten zwei speziellen Typen von Konstruktoren: der Default-Konstruktor und der Copy-Konstruktor. Ein Konstruktor, der ohne Parameter aufgerufen werden kann, wird als Default-Konstruktor bezeichnet. Ein Konstruktor, der mit einem Parameter vom gleichen Typ aufgerufen um das Objekt zu initialisieren werden kann, wird als Copy-Konstruktor bezeichnet. Der Copy-Konstruktor ist wie folgt definiert:

```
class_name(const class_name& object);
```

Das reservierte Wort *const* zeigt, dass der Parameter nicht verändert werden kann. Klassen können auch Attribute von anderen Klassen-Typen besitzen. Für die folgende Klasse:

```
class class_name {
      class_name_1 ob_1;
      class_name_2 ob_2;
      ...
      class_name_n ob_n;
      ...
};
```

der Konstruktor hat die folgende Deklaration:

```
class_name(argument_list):
           ob_1(l_arg_1), ob_2(l_arg_2), ..., ob_n(l_arg_n)
```

wobei argument_list und l_arg_i beschreiben die Argumente für die Konstruktoren der Objekte *class_name* und *ob_i*.

Von der `ob_1(l_arg_1), ob_2(l_arg_2), ..., ob_n(l_arg_n)` kann man Objekte in die folgenden Fälle entfernen: das Objekt hat keinen Konstruktor, der Objekt wird durch einen Default-Konstruktor erzeugt, der Konstruktor des Objekts besitzt nur Default-Parameter.

Falls Attribute einer Klasse selbst vom Klassentyp sind, werden bei einer Objekterzeugungzuerst die Konstruktoren der Attribute aufgerufen.

File `pair.cpp`:

```
#include <iostream>
#include "person.h"

using namespace std;

class pair {
      person husband;
      person wife;
public:
      pair() {} //implicit constructor definition
```

```
                  //the implicit constructors }
                  //for objects husband and wife are called

      pair(person& ahusband, person& awife);
      pair(char* lname_husband, char* fname_husband,
            char* lname_wife, char* fname_wife):
            husband(lname_husband, fname_husband),
            wife(lname_wife, fname_wife) {}

      void print();
};

inline pair::pair(person& ahusband, person& awife):
      husband(ahusband), wife(awife) { }

void pair::print() {
      cout << "husband: ";
      husband.print();
      cout << "wife: ";
      wife.print();
}

int main() {
      person A("Pop", "Ion");
      person B("Popa", "Ioana");
      pair AB(A, B);
      AB.print();
      pair CD("C","C","D","D");
      CD.print();
      pair EF;
      EF.print();
      return 0;
}
```

Die Parameter *husband* und *wife* sind als Objektreferenz deklariert. Hätten sie als Person-Typ deklariert, wird der Copy-Konstruktor 4 times aufgerufen werden:

```
      pair AB(A, B);
```

In diesen Situationen sind temporäre Objekte durch den Copy-Konstruktor (zwei Mal in diesem Fall) erzeugt. Und danach werden die Konstruktoren der Attribute aufgerufen (noch zwei Mal).

1.2.1.6 Der Destruktor

Ein Destruktor ist das Gegenstück zum Konstruktor. Er ist eine weitere spezielle Methode einer Klasse, der automatisch bei Beendigung der Lebensdauer eines Objekts der Klasse aufgerufen wird. Der Destruktor der globalen Objekte wird am ende der Main-Funktion durch den Aufruf von exit() automatisch aufgerufen. Somit ist der Aufruf von exit() innerhalb des Destruktor nicht empfohlen. Für lokale Objekte wird der Destruktor zu dem Zeitpunkt aufgerufen, an dem das lokale Objekt gelöscht wird. In der Regel ist dies die Stelle, an der eine geschweifte Klammer zu steht. Für dynamische Objekte wird der Destruktor mit dem Operator indirekt aufgerufen. Es gibt noch eine explizite Möglichkeit den Konstruktor mit dem Klassenname und dem Scope Resolution Operator zu aufrufen.

Der Destruktorname ist der Klassenname mit vorangestelltem ~. Wie der Konstruktor es hat kein Funktionstyp (auch nicht void). Manche Situationen wird im Folgenden beschrieben:

File `destruct.cpp`:

```
#include <iostream>
#include <cstring>

using namespace std;

class write { //write on stdout what it does.
      char* name;
public:
      write(char* n);
      ~write();
};

write::write(char* n) {
      name = new char[strlen(n)+1];
      strcpy(name, n);
      cout << "Created object: " << name << '\n';
}

write::~write() {
      cout << "Destroyed object: " << name << '\n';
      delete name;
}

void function() {
      cout << "Call function" << '\n';
      write local("Local");
}

write global("Global");

int main() {
      write* dynamic = new write("Dynamic");
      function();
```

```
	cout << "In main" << '\n';
	delete dynamic;
	return 0;
}
```

1.3 Beziehungen zwischen Klassen

1.3.1 Theorie

Der Benutzer eines Objekts braucht seinen genauen Aufbau nicht zu kennen. Ihm müssen lediglich die Methoden, die er für eine Interaktion mit dem Objekt benötigt. Von der internen Darstellung der den jeweiligen Objektzustand festlegenden Daten braucht er dagegen keinerlei Kenntnis zu haben. Somit sind die Daten nach außen *gekapselt*.

Die Vererbung ist eine Weitergabe von Daten und Funktionen eines Objekts an ein anderes Objekt. Statt eine Klasse völlig neu zu implementieren, wird auf bereits Vorhandenes zurückgegriffen und nur das was neu hinzukommt oder sich ändert als Ergänzung definiert. Durch *Vererbung* lassen sich also Gemeinsamkeiten zwischen Klassen erfassen und in hierarchische Beziehungen einordnen. Die abgeleitete Klasse erbt die Daten und Methoden der Basisklasse und kann neue Daten und Methoden besitzen. Eine neue Klasse wird auch mehreren bereits definierten Klassen abgeleitet (*multiple inheritance*).

Polymorphie ist die Verwendung des gleichen Namens für unterschiedliche Dinge. In der OOP ermöglicht es, dass verschiedenartige Objekte unter einem gemeinsamen Basisklasse betrachtet und bearbeitet werden können. In der Basisklasse definierte Funktionen können in abgeleiteten Klassen mit dem gleichen Namen und der gleichen Parameterliste erneut definiert werden (überschrieben). Dadurch wird das durch den Aufruf einer derartigen Funktion bewirkte Verhalten eines Objekts in der abgeleiteten Klasse abgeändert. Der gleiche Funktionsname kann somit zur Spezifizierung einer Gruppe ähnlicher aber doch unterschiedlicher Operationen Methoden verwendet werden.

1.3.2 Vererbung

Ableitung einer Klasse:

```
class name_of_derived_class : list_of_base_classes {
	//new attributes and methods
};
```

wobei `list_of_base_classes` ist:

$elem_1$, $elem_2$, ..., $elem_n$

und $elem_i$ für $1 \leq i \leq n$ kann sein:

```
public base_class_i
```

or

```
protected base_class_i
```

or

```
private base_class_i
```

In der Abstammungsliste einer abgeleiteten Klasse kann für jede Basisklasse ein Zugriffs-Specifier angegeben werden (Tabelle 1). Falls für eine Basisklasse kein Vererbungs-Zugriffs-Specifier angegeben ist, gilt private als Default.

Zugriff von der Basisklasse	**Vererbungs-Zugriffs-Specifier**	**Zugriff von der abgeleiteten Klass**
public	public	public
protected	public	protected
private	public	unerreichbar
public	protected	protected
protected	protected	protected
private	protected	unerreichbar
public	private	private
protected	private	private
private	private	unerreichbar

Tabelle 1. Zugriff von der abgeleiteten Klasse

Zu private Komponenten der Basisklasse hat die abgeleitete Klasse grundsätzlich keinen direkten Zugriff. Protected bewirkt, dass nur abgeleitete Klassen Zugriff auf die Komponente bekommen. Public ist allerdings meist nur für Methoden gedacht, weil öffentliche Attribute gegen das Konzept der Datenkapselung verstoßen. Beachten Sie, dass friend-Beziehungen nicht vererbt werden.

1.3.3 Virtuelle Funktionen

Polymorphismus ermöglicht es, dass mehrere Methoden den gleichen Namen haben können. Die Auswahl der über einen Funktionsnamen tatsächlich aufgerufenen Funktion, d.h. die Zuordnung einer Methode (Member-Funktion) zu einem Funktionsaufruf wird in der

OOP als Binden bezeichnet. Das folgende Beispiel betrachten, in dem base eine Basisklasse ist, und derived eine geleitete Klasse. Die Basisklasse bietet folgende Methoden an: *method_1* (die *method_2* aufruft) und method_2. In der abgeleiteten Klasse wird nur *method_1* überschrieben. Ein Objekt der abgeleiteten Klasse wird In der Main-Methode erstellt, und die Methode *method_2* aufgerufen wird. Eine mögliche Implementation ist:

File `virtual1.cpp`:

```
#include <iostream>
using namespace std;

class base {
public:
      void method_1();
      void method_2();
};

class derived : public base {
public:
      void method_1();
};

void base::method_1() {
      cout << "Method method_1 of the"
            << " base class is called"
            << endl;
}

void base::method_2() {
      cout << "Method method_2 of the"
            << " base class is called"
            << endl;

      method_1();
}

void derived::method_1() {
      cout << "Method method_1 of the"
            << " derived class is called"
      << endl;
}

int main() {
      derived D;
      D.method_2();
}
```

Die Ausgabe des Codes ist:

```
Method method_2 of the base class is called
Method method_1 of the base class is called
```

Aber die Ausgabe ist nicht das, was wir uns erwartet haben, weil die Methode der Basisklasse statt die Methode der abgeleiteten Klasse aufgerufen wird. In diesem Fall wird die Auswahl der Methode bereits zur Compilezeit erfolgt.

Virtuelle Funktionen können dieses Problem lösen. Wenn eine Methode virtuelle definiert wird, erfolgt die Zuordnung erst zur Laufzeit. Die Startadresse der tatsächlich aufgerufenen Funktion wird nicht vom Compiler ermittelt, sondern erst zur Laufzeit in Abhängigkeit vom jeweiligen Ziel-Objekt festgelegt. Diese Eigenschaft wird Spätes Binden genannt. Wenn die Zuordnung bereits zur Compilezeit erfolgt, sprechen wir über Frühes Binden.

Doch obwohl die Methode *method_1* durch die abgeleitete Klasse umdefiniert worden ist, die Methode der Basisklasse durch frühes Binden aufgerufen wird. Um dieses Verhalten zu ändern, kann man die Methode *method_1* virtuelle machen.

In C++ kann man das reservierte Wort *virtual* benutzen, um eine Methode virtuelle zu machen. Zum Beispiel, die Basisklasse im vorherigen Beispielsfall kann werden:

```
class base {
public:
      virtual void method_1();
      void method_2();
};
```

Die Ausgabe des Codes ist:

```
Method method_2 of the base class is called
Method method_1 of the derived class is called.
```

Wir stellen im Folgenden ein anderes Beispiel, um die Notwendigkeit der virtuellen Funktionen zu betonen. Wir definieren die Klasse *fraction* mit Konstruktor der einen Bruch mit dem Zähler *z* und dem Nenner *n* erzeugt. Die Klasse soll weiters über Methoden multiply (multipliziert den aktuellen Bruch mit einem anderen), und product (berechnet die Multiplikation zwischen zwei Brüche) verfügen. Eine weitere Methode soll den Bruch auf dem Bildschirm ausgeben. Eine Klasse *fraction_write* erweitert fraction. Es überschreibt die Methode product, um das Ergebnis auf dem Bildschirm auszugeben.

File `fvirt1.cpp`:

```
#include <iostream>
using namespace std;

class fraction {
protected:
      int numerator;
      int denominator;
public:
```

```
      fraction(int numerator1 = 0, int denominator1 = 1);
      fraction product(fraction& r); //computes the product of two
      //fractions, but does not simplify fraction&

      multiply(fraction& r);
      void display();
};
fraction::fraction(int numerator1, int denominator1) {
      numerator = numerator1;
      denominator = denominator1;
}

fraction fraction::product(fraction& r) {
      return fraction(numerator * r.numerator, denominator *
r.denominator);
}

fraction& fraction::multiply(fraction& q) {
      *this = this->product(q); return *this;
}

void fraction::display() {
      if ( denominator )
            cout << numerator << " / " << denominator;
      else
            cerr << "Incorrect fraction";
}

class fraction_write: public fraction{
public:
      fraction_write(int numerator1 = 0, int denominator = 1 );
      fraction product(fraction& r);
};

inline     fraction_write::fraction_write(int     numerator1,     int
denominator1):
fraction(numerator1, denominator1) { }

fraction fractie_write::product(fraction& q) {
      fraction r = fraction(*this).product(q);
      cout << "(";
      this->display();
      cout << ") * (";
      q.display();
      cout << ") = ";
      r.display();
      cout << endl;
      return r;
```

```
}

int main() {
      fraction p(3,4), q(5,2), r;
      r = p.multiply(q);
      p.display();
      cout << endl;
      r.display();
      cout << endl;
      fraction_write p1(3,4), q1(5,2);
      fraction r1, r2;
      r1 = p1.product(q1);
      r2 = p1.multiply(q1);
      p1.display();
      cout << endl;
      r1.display();
      cout << endl;
      r2.display();
      cout << endl;
      return 0;
}
```

Die Ausgabe des Codes ist:

```
15 / 8
15 / 8
(3 / 4) * (5 / 2) = 15 / 8
15 / 8
15 / 8
15 / 8
```

Wir beobachten, dass die Ausgabe nicht das, was wir uns erwartet haben. Die Multiplikation nur einmal, nämlich durch `r1 = p1.product(q1)`, angezeigt ist. Der ausdruck `r2 = p1.multiply(q1)` zeige die Multiplikation nicht an, weil die Methode *multiply* die abgeleitete Klasse nicht umdefiniert ist und die Auswahl der Methode erfolgt bereits zur Compilezeit.

Um dieses Verhalten zu ändern, kann man wie das andere Beispiel die Methode *product* virtuelle machen. Die umdefinierte Basisklasse ist:

```
class fraction {
protected:
      int numerator;
      int denominator;
public:
fraction(int numerator1 = 0, int denominator = 1);
      virtual   fraction   product(fraction&   r);   //computes   the
      //product of two fractions, but does not simplify
```

```
      fraction& multiply(fraction& r); void display();
};
```

Die Ausgabe des Codes ist:

```
15 / 8
15 / 8
(3 / 4) * (5 / 2) = 15 / 8
(3 / 4) * (5 / 2) = 15 / 8
15 / 8
15 / 8
15 / 8
```

Eine virtuelle Funktion muss in der abgeleiteten Klasse nicht neu definiert werden. In diesem Fall erbt die abgeleitete Klasse die virtuelle Funktion der Basisklasse. Enthält eine Klasse virtuelle Funktionen, so wird für diese Klasse vom Compiler eine VirtuelleMethoden-Tabelle angelegt. In dieser sind die Anfangsadressen aller virtueller Funktionen zusammengefaßt. Aus diesem Grund dauert der Aufruf einer virtuellen Funktion etwas länger als der Aufruf einer nicht-virtuellen Funktion.

1.3.4 Abstrakte Klasse

In so eine komplizierte Kontext von Hierarchie Klassen, die Hauptklasse kann einige Eigenschaften haben, aber die Implementierung von Eigenschaften wird in jede abgeleitete Klasse definiert. Die Abbildung 1.3 wird als Beispiel genommen.

Man kann einige Eigenschaften von einer abgeleiteten Klasse finden. Zum Beispiel: Durchschnittsgewicht, Lebensdauer und Geschwindigkeit. Diese Eigenschaften wird mit verschiedene Methoden umgesetzt. Im prinzip, Durchschnittsgewicht, Lebensdauer und Geschwindigkeit sind Eigenschaften auch für Animal Klasse, aber sind zu schwierig zu bestimmen und nicht so wichtig für uns in so ein gemeinsames Kontext. Doch für eine einheitliche Behandlung, wird es besser wenn die drei Methoden in Hauptklasse definiert sind und werden sie in jede abgeleitete Klasse umgesetzt. Pure virtual method ist um diesem Zweck zu definieren.

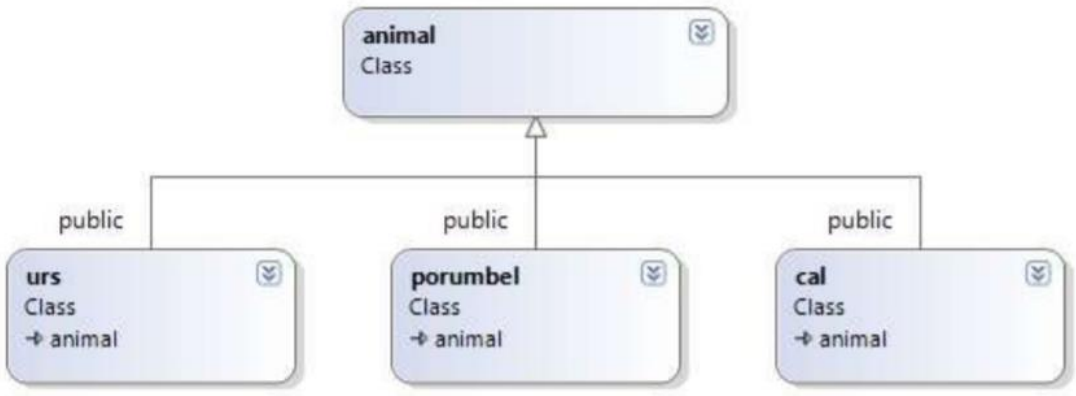


Abbildung 1.3 Animals Klassenhierarchie

Eine sogenannte pure virtual Methode ist eine Methode, die in eine Klasse definiert ist aber in andere Klasse umgesetzt. Es **muss** in eine abgeleitete Klasse umgesetzt. Eine pure virtuelle Methode nimmt den Präfix **virtual** und den Header endet mit = 0. Der Auswahl, von welche Implementierung zu nehmen, werden während der Ausführung das Programm gemacht.
Die Klassen, die enthält mindestens eine pure virtuellen Methode sind abstrakte Klassen genannt. Abstrakte Klassen enthalten Methoden, die keine Implementierung da haben. Deswegen, kann man nicht eine neue Objekt von abstrakte Klasse definieren. Falls eine von pure virtuellen Methoden keine Implementierung in die abgeleitete Klasse habe, dann die abgeleitete Klasse ist als abstrakte Klasse betrachtet. In diesem Falls ist es auch **nicht möglich** um eine neue Objekt zu erstellen.

Nehmen wir die obige Beispiel und schreiben wir ein Programm, die prüft ob ein dove, ein bear, oder ein horse fat oder skinny, fast oder slow, und old oder young ist. Das Ergebnis wird bei eine Methode von Animal Klasse, die nicht in abgeleitete Klasse überschrieben ist, geschrieben.
File `abstract1.cpp`**:**

```
#include <iostream>
using namespace std;

class animal {
protected:
      double weight; // kg 28
      double age; // years
      double speed; // km / h
public:
      animal( double w, double a, double s);
      virtual double average_weight() = 0;
      virtual double average_lifespan() = 0;
      virtual double average_speed() = 0;
      int fat() { return weight > average_weight(); }
      int fast() { return speed > average_speed(); }
      int young() { return 2 * age < average_lifespan(); }
      void display();
};

animal::animal( double w, double a, double s) {
weight = w;
age = a;
speed = s;
}

void animal::display() {
cout << ( fat() ? "fat, " : "skinny, " );
cout << ( young() ? "young, " : "old, " );
cout << ( fast() ? "fast" : "slow" ) << endl;
```

```
}

class dove : public animal {
public:
      dove( double w, double a, double s): animal(w, a, s) {}
      double average_weight() { return 0.5; }
      double average_lifespan() { return 6; }
      double average_speed() { return 90; }
};

class bear: public animal {
public:
      bear( double w, double a, double s): animal(w, a, s) {}
      double average_weight() { return 450; }
      double average_lifespan() { return 43; }
      double average_speed() { return 40; }
};

class horse: public animal {
public:
      horse( double w, double a, double s): animal(w, a, s) {}
      double average_weight() { return 1000; }
      double average_lifespan() { return 36; }
      double average_speed() { return 60; }
};

int main() {
      dove d(0.6, 1, 80);
      bear b(500, 40, 46);
      horse h(900, 8, 70);
      d.display();
      b.display();
      h.display();
      return 0;
}
```

Die abstrakte Klasse Animal setzt einige Methoden um. Der Vorteil einer solchen Klasse zu implementieren, ist dass die drei abgeleitete Klassen kann einfach die Methode ohne andere Änderungen ererben. Das führt zu Wiederverwendung von Code.

1.3.5 Interfaces

Im C++ Programmierung Sprache existiert keine Interfaces, die schon in Java oder C# existiert. Für diesen Zweck zu definieren, kann man eine Abstrakte Klasse definieren, die hat nur pure virtual Methoden. Offensichtlich ist es dass in diesem Fall, keine Attributen innerhalb die Klasse definieren **muss**.
Abstract Klasse Animal enthält beide Sachen: sowohl Attributen als auch non virtuellen Methoden. Deswegen, kann man Animal Klasse nicht sowie eine Interface betrachten.

Letztendlich, wird es unten eine Abstrakte Klasse Vehicle definiert, die nur pure virtuellen Methode enthält. Doch, definiert man noch zwei abgeleitete Klassen von Vehicle.
File `vehicle.cpp`:

```
#include <iostream>
using namespace std;

class Vehicle {
public:
      virtual void Start() = 0;
      virtual void Stop() = 0;
      virtual void Go(int km) = 0;
      virtual void Stand (int min) = 0;
};

class Bicycle : public Vehicle {
public:
      void Start();
      void Stop();
      void Go(int km);
      void Stand(int min);
};

void Bicycle::Start() {
      cout << "The bicycle starts." << endl;
}

void Bicycle::Stop() {
      cout << "The bicycle stops." << endl;
}

void Bicycle::Go(int km) {
      cout << "The bicycle goes " << km << " kilometers." << endl;
}

void Bicycle::Stand(int min) {
      cout << "The bicycle stands " << min << " minutes." << endl;
}

class Car : public Vehicle {
public:
      void Start();
      void Stop();
      void Go(int km);
      void Stand(int min);
};

void Car::Start() {
```

```
      cout << "The car starts." << endl;
}

void Car::Stop() {
      cout << "The car stops." << endl;
}

void Car::Go(int km) {
      cout << "The car goes " << km << " kilometers." << endl;
}

void Car::Stand(int min) {
      cout << "The car stands " << min << " minutes." << endl;
}

void Path(Vehicle *v) {
      v->Start();
      v->Go(3);
      v->Stand(2);
      v->Go(2);
      v->Stop();
}

int main() {
      Vehicle *b = new Bicycle;
      Path(b);
      Vehicle *c = new Car;
      Path(c);
      delete b;
      delete c;
}
```

In die *main* Methode, zwei dynamische Objekten vom Typ *Bicycle* und *Car* sind so definiert dass mit den Anruf von *Path* Methode, wird es zwei verschiedene Ergebnisse liefern obwohl als formal Parameter nur eine Pointer zur Abstrakt Klasse *Vehicle* ist.

1.4 Klassendiagramme und UML Interaktionen zwischen Objekten: Pakete, Klassen und Interfaces. Beziehungen zwischen Klassen und Interfaces. Objekte. Messages

Die Unified Modeling Language (UML) dient primär der Erstellung von objektorientierten Analyse- und Entwurfsmodellen im Rahmen der Entwicklung von betrieblichen Informationssystemen (kann aber auch für andere Informationssystemen benutzt werden).

UML definiert eine Menge von Modellierungsobjekten und graphischen Notationen die zu unterschiedlichen Elemente gehören.

Dieser Abschnitt enthält ein Teil der Basiselemente, die benutzt werden können um die Struktur und das Verhalten eines Objektes in einem objektorientierten Informationssystem zu beschreiben – *Klassendiagramme* und *Interaktionsdiagramm*. Diese Elemente entsprechen der Selektion von [30], Kapitel 3 und 4.

Bevor wir diese Elemente vorstellen, wollen wir den Kontext vorstellen, in dem diese Elemente für Software Entwicklung benutzt werden. Die Hauptfragen, die beantwortet werden müssen, sind: (A) **welche Typen von Modellen** brauchen wir, (B) **wann sollen wir diese Modelle aufbauen**, abhängig von dem Entwicklungsprozess und (C) welche ist die **Beziehung zwischen den Modellen und dem geschriebenen Code.**

Um diese Fragen zu beantworten, werden wir ein paar Beispiele geben bezüglich einer Anwendung, die von einem Kassierer benutzt wird um die Verkäufe an einem Verkaufspunkt in dem Laden aufzuzeichnen. Die Anwendung heißt **POS** (Point of Sale) und implementiert einen einzigen Use Case, Aufzeichnung der Verkäufe.

A. Typen von Modellen

Bezüglich der Modellen Typen ist es angemessen die Konzepte eingeführt von Modellgetriebene Architektur zu benutzen, nämlich **Model-Driven Architecture – MDA** [31]. In den folgenden Abschnitten beschreiben wir die Modelle eingeführt von dem MDA Handbuch.

CIM – Berechnungs unabhängige Modelle (Computation Independent Models). Diese Modelle beschreiben **was** das System tut und nicht **wie dieses Verhalten bereitgestellt wird.** Hier werden die Anforderungen einer Applikation in einer formalen Spezifikation zusammengefasst, die unabhängig von der Realisierung des Systems ist. Das Ziel dabei ist, die Aufgaben eines Systems in dessen Umfeld zu beschreiben, so dass **CIM** auch häufig als **Domänenmodell** (**Domain Models)** oder **Geschäftsmodell** (**Business Models)** bezeichnet wird. Aus der Perspektive der Struktur, werden Klassendiagrame dafür benutzt, **Domain Konzepte** zu definieren. Die Interaktionsdiagramme werden selten für **CIM** benutzt. Um das gewünschte Verhalten des Systems zu beschreiben werden andere Elemente benutzt, so wie Use Cases, Business Prozesse, usw. – aber diese werden in diesem Abschnitt nicht besprochen.

Das erste Modell aus der Abbildung 1.4 stellt ein Teil des konzeptuellen Modells für die POS Anwendung dar. Das Modell illustriert Konzepte, die von dem Benutzer benutzt werden. Diese Konzepte werden dann zusammen mit anderen Modelling Elemente benutzt, um das gewünschte Verhalten zu beschreiben.

PIM – Plattform unabhängige Modelle (Platform Independent Models). Diese Modelle beschreiben **wie das System funktioniert**, unabhängig von der Technologie der Implementierung oder Plattform. **PIM** baut auf dem CIM auf und beschreibt die Struktur und das Verhalten des geplanten Software-Systems. Auf diesem Niveau, werden Architekturelemente eingeführt, und die Klassen- und Interaktionsdiagramme zwischen den Objekten sind zwei wichtige Tools um das System detailliert zu beschreiben (**detailliertes Design**). Man kann auch andere Elemente verwenden, die aber hier nicht beschrieben werden – z.B. Kollaborationsdiagramme,usw. Also enthält das PIM nicht nur statischen Informationen (z.B. über die Darstellung im Use Case Diagramm bzw. Klassendiagramm), sondern auch die dynamischen Aspekte (z.B. über Aktivitätsdiagramm, Sequenzdiagramm und Zustandsdiagramm).

Das zweite Modell aus der Abbildung 1.4 stellt ein Teil des PIM Modells für POS dar. Beide Modelle, CIM und PIM, enthalten nur UML Strukturen (z.B. Datentypen definiert mit der UML Spezifikation) und mögliche Erweiterungen dieser Sprache (Plattform unabhängig).

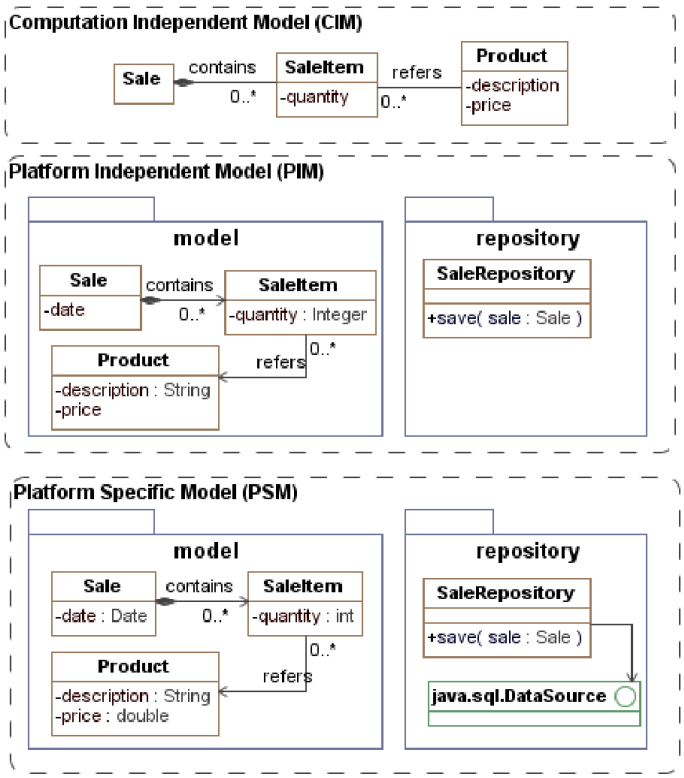


Abbildung 1.4 Modellen Typen

PSM – Plattform ausgerichteten Modelle/Plattform spezifische Modelle (Platform Specific Models). Ein PSM stellt eine Spezialisierung des PIM dar, und zwar wird das allgemeine PIM um solche Aspekte erweitert, die rein plattformspezifisch sind. Der Architekt kann die Erstellung eines solches Modells entscheiden, um unterschiedliche Elemente der ausgewählten Plattform darzustellen, wie z.B. benutzte Datentypen. Die Klassen- und Interaktionsdiagramme werden für diese Modelle auch benutzt.

Das letzte Modell aus der Abbildung 1.4 stellt eine Transformation des PIM Modells für eine Java Implementierung dar. Die Datentypen aus diesem Modell sind Java Typen (z.B. String, double), und das Modell enthält sogar ein Datentyp aus dem Paket java.sql.

Man definiert solche Modelle um letztendlich durch eine Modell-zu-Code-Transformation den Programmcode zu generieren für die ausgewählten Plattformen. Der Code kann aus dem PIM oder PSM Modell generiert werden. In der Transformation werden Beziehungen zwischen den Elementen des Modells und den Elementen der ausgewählten Plattform benutzt. Die Generierung von Code aus einem Plattform-spezifischen Modell soll im Vorgehen der Model Driven Architecture zur Steigerung der Qualität und Verbesserung der Wartbarkeit beitragen.

B. Entwicklungsprozessen und CASE Tools

Unterschiedliche Entwicklungsprozesse deuten die Verwendung unterschiedlichen Typen von Modellen an unterschiedlichen Entwicklungsphasen an.

Das **Model Driven Entwicklungsprozess** entspricht im Allgemeinen dem Handbuch [31] und erstellt PIM Modelle, die optional von den CIM Modellen abgeleitet wurden. Dann

wird von den CIM Modellen Code generiert, optional mithilfe von PSM Zwischenmodellen. Während diesen Entwicklungsprozessen muss man Design Tools (CASE – Computer Aided Software Engineering) benutzen, welche die Infrastruktur der Modell-Transformation unterstützen. Beispiele davon sind die Model Driven Prozesse für Service Oriented Applikationen, welche Plattform unabhängige Sprachen/Erweiterungen von UML, sowie SoaML[1] benutzt.

Es gibt **Modellgetriebene Prozesse, welche keine PIM Modelle, sondern direkt PSM Modelle benutzen**. Diese beziehen sich auf Spezifikationen für unterschiedliche Plattformen, sowie Component Based Architekturen welche Dienstleister (service provider) sind, SCA[2].

Entwicklungsprozesse die mehr anspruchsvoll sind, sowie RUP[3] (für große Systeme), empfehlen die Verwendung aller Modelle, **CIM, PIM** und **PSM**, zusammen mit der Verwendung von CASE Tools.

Das Agile Entwicklungsprozess, sowie Test Driven Entwicklung[4] oder Model Driven Agile Entwicklung[5], empfehlen im Allgemeinen die Verwendung von CASE Tools nicht, aber sie empfehlen die Verwendung von Modellen bevor man Code schreibt. **Die Modelle sind eigentlich Skizzen** (auf dem Papier oder auf einer Tafel) und werde dafür verwendet, Ideen über das System Design darzustellen.

Unabhängig davon, welcher Entwicklungsprozess verwendet wird, erlauben die meisten modernen Entwicklungstools **eine direkte Synchronisation zwischen dem geschriebenen Code und den entsprechenden Modellen.** Diese Synchronisation findet eigentlich zwischen dem Code und den PSM Modellen statt. Zum Beispiel, ein CASE Tool welcher die Modelle mit dem geschriebenen Java Code synchronisiert, beeinflusst eigentlich nur die PSM Modelle für die Java Plattform.

Eine letzte und neue Kategorie von Entwicklungsprozessen, welche die Verwendung von PIM Modellen und die direkte und vollständige Generierung von Code vorschlägt, ist die Kategorie von Prozessen basierend auf **executable (ausführbare) Modellen.** Heutzutage, ist die Annahme des Standards der ausführbaren UML Modellen (fUML – Foundational UML)[6] fast zum Schluss gekommen. Gemäß diesen Prozessen, erwarten wir **einen neuen Entwicklungsstil** wo man nur Modelle erstellt, die dann benutzt werden um Code **in einer textuellen Sprache[7], definiert mithilfe der Elemente aus der Modellen**, zu generieren. In dieser Art, werden die PSM Modellen und der Code geschrieben in Sprachen wie Java, C++ oder C# automatisch, von CASE Tools, generiert.

C. Die Beziehung zwischen den Modellen und dem Code

Wie man bei (A) und (B) gesehen hat, sind die Beziehungen zwischen den Modellen und

---

[1] OMG. *Service Oriented Architecture Modeling Language*, 2009. http://www.omg.org/spec/SoaML/

[2] Open SOA. *Service Component Architecture Specifications*, 2007. http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications

[3] IBM. *IBM Rational Unified Process*, 2007. http://www-01.ibm.com/software/awdtools/rup/

[4] Beck, K. *Test-Driven Development by Example*, Addison Wesley, 2003.

[5] Ambler, S.W. *Agile Model Driven Development (AMDD): The Key to Scaling Agile Software Development*, 2008. http://www.agilemodeling.com/essays/amdd.htm

[6] OMG. Semantics Of A Foundational Subset For Executable UML Models (FUML), 2010. http://www.omg.org/spec/FUML/

[7] OMG. Concrete Syntax For UML Action Language (Action Language For Foundational UML - ALF), 2010. http://www.omg.org/spec/ALF/

dem Code wichtig. Wenn man Code von den PIM Modellen generiert, oder wenn man ein CASE Tool benutzt, der die PSM Modellen mit dem Code synchronisiert, dann ist es wichtig zu wissen welche Typen von Elementen von den Modellen generiert werden. Auch wenn wir agile arbeiten und Skizzen benutzen (ohne ein CASE Tool zu benutzen), treten dieselben Probleme auf.

Bezüglich der ausführbaren (executable) Modellen, die vorher erwähnt wurden (und welche auf dem PIM Ebene sind), werden wir in den nächsten Abschnitten nur die **Beziehungen zwischen PIM Modellen und Sprachen wie C++, Java und C#** besprechen. Die PSM Modellen enthalten zusätzlich zu den PIM Modellen auch Datentypen spezifisch für entsprechende Sprachen. Folgendermaßen sind die Beziehungen zwischen den PSM Modellen und dem Code dieselben, aber die Modelle enthalten zusätzlich auch UML Erweiterungen entsprechend zu den spezifischen Datentypen.

1.4.1 Klassendiagramme

Ein **Diagramm** ist eine graphische Darstellung (meistens 2D) von Elementen eines Modells. Das **Klassendiagramm** stellt die Typen von Objekten, die in einem System benutzt werden, sowie die Beziehungen dazwischen dar. Die strukturellen Elemente, die in diesem Abschnitt beschrieben werden, sind (a) **Typen von Objekten**: Klasse, Interface, Aufzählungen (enumerations); (b) **Gruppierungen von Elementen** mithilfe von Paketen und (c) **Beziehungen zwischen diesen Elementen:** Assoziation, Generalisierung/Verallgemeinerung, Realisierungen (realization) und Abhängigkeiten.

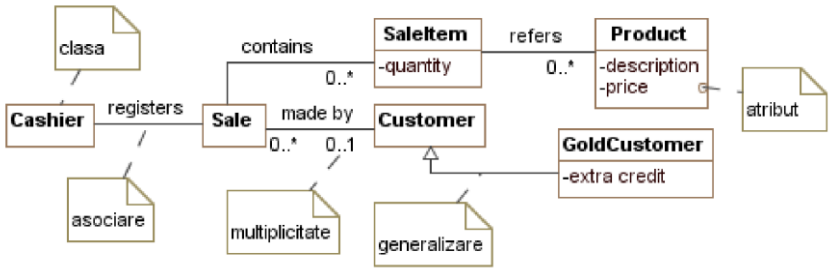


Abbildung 1.5 Konzeptuellen Modell

Abbildung 1.5 stellt ein anfängliches konzeptuelles Modell für POS dar. Die Klassen werden dafür benutzt, die Konzepte eines Domains zu identifizieren. Wenn es nicht relevant ist, dann wird der Teil für die Attribute der Klasse verborgen. Die Eigenschaften der Klassen werden durch Attributen und Assoziationen definiert, und die Datentypen für die Attribute werden nicht angegeben.

Die konzeptuellen Modelle gehören zu dem CIM Typ und werden dafür benutzt, PIM Modelle zu generieren. Sowie die CIM Modelle, dürfen diese keine Details über die Darstellung der Attribute enthalten. Wenn das Entwicklungsprozess kein CIM Modell braucht, sondern ein PIM oder PSM Modell, dann ist das Modell aus der Abbildung 1.5 ein PIM oder ein unvollständiges PSM Modell.

Im Zusammenhang mit PIM, wird die Architektur eines Systems aus strukturellen Perspektive mithilfe von Pakete (hierarchisch organisiert) und Abhängigkeiten dazwischen beschrieben – siehe Abbildung 1.6 für POS. Die Pakete werden mit zusammenhängenden Verantwortungen definiert, so wie die Interaktion mit dem Benutzer (*UI*), Fassade über die

Domäne (*service*), Entitäten (*model*) und Repository von Objekten oder Datenzugang-Objekten (*repository, inmemory repository*).

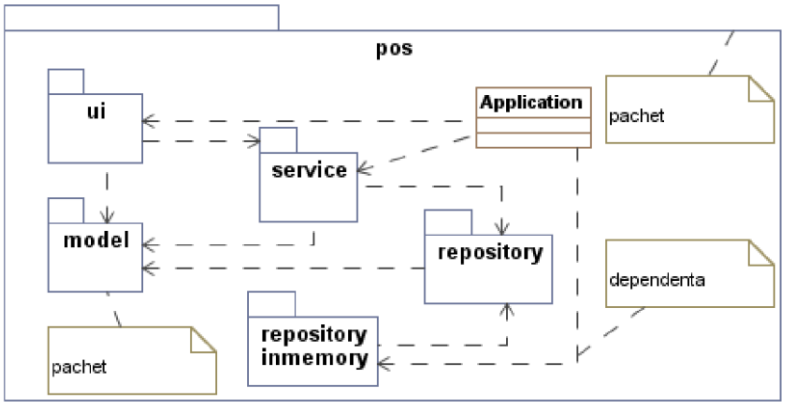


Abbildung 1.6 Geschichtete (stratified, layered) Architektur

Die Hauptsorge bei der Entscheidung der Architektur eines Systems ist alle SOLID[8,9] objektorientierten Prinzipien zu folgen. Die Pakete aus der Abbildung 1.6 sind nach dem Single-Responsibility-Prinzip entworfen, wonach ein Objekt nur eine Verantwortung haben soll und Objekte mit zusammenhängenden Verantwortungen logisch gruppiert werden müssen.

Die Abhängigkeiten zwischen zwei Software Elementen (A hängt von B ab) weisen darauf hin, dass, bei der Änderung des Objektes B, wahrscheinlich auch A geändert werden muss. Die Abhängigkeiten zwischen den Paketen in der Abbildung 1.6 folgen die Empfehlungen der geschichteten Architektur, d.h. die Elemente aus den höheren Schichten hängen von den unteren Schichten ab. Zum Beispiel, hängt *UI* von *service* und *model* ab, *service* hängt von *model* und *repository* ab, aber *service* hängt nicht von der konkreten Implementierung für das Repository ab, nämlich *repository inmemory.* Die Abhängigkeiten umzudrehen folgt auch einem SOLID Prinzip, nämlich Dependency-Inversion-Prinzip. Abbildung 1.7 stellt die Details dar bezüglich der Inversion der Abhängigkeit zwischen *service* und *repository inmemory.* Damit *StoreService* nicht mehr abhängig von der konkreten Implementierung *InmemorySaleRepository* ist, wurde die Interface *SaleRepository* eingeführt, welche die zwei Elemente entkoppelt. Eigentlich, abstrahiert *SaleRepository* den Zugang zu *Sale* Objekten und erlaubt damit das *repository inmemory* System mit einer anderen Implementierung zu ersetzen, ohne dadurch andere Pakete aus dem System zu beeinflussen.

[8] Robert C. Martin. *Design Principles and Design Patterns*, 2004. http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[9] SOLID Design Principles: *Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion*, http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

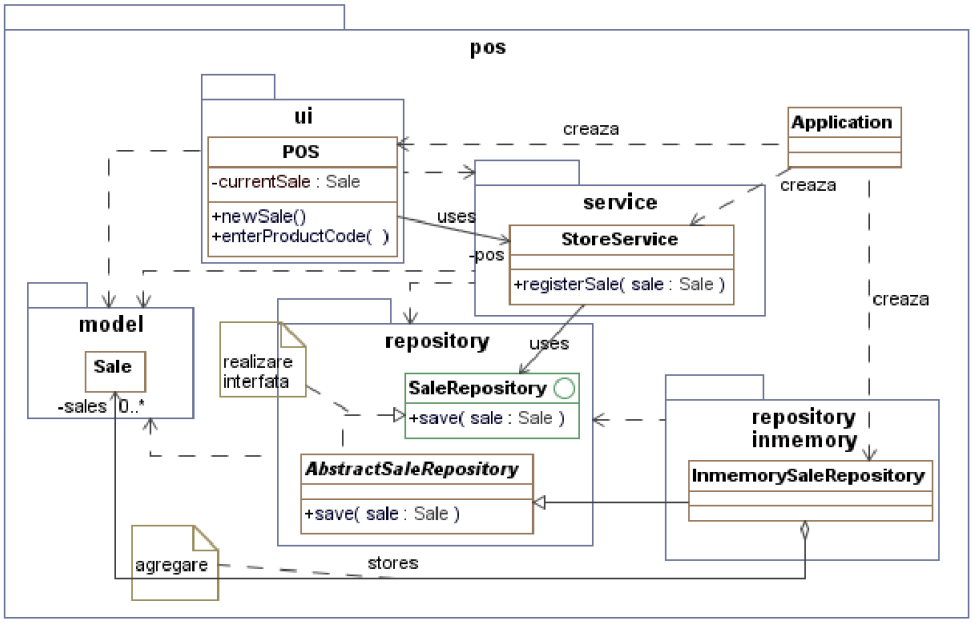


Abbildung 1.7 Geschichtete Architektur - Erklärung der Abhängigkeiten

Auf PIM oder PSM Ebene, werden Klassendiagramme dafür benutzt, Entitäten und die Beziehungen dazwischen zu verfeinern – siehe Abbildung 1.8. Diese Elemente werden in den folgenden Abschnitten besprochen.

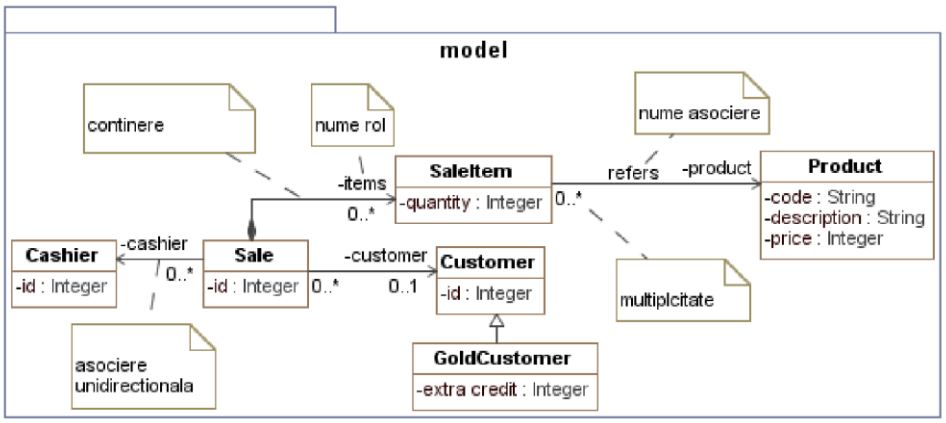


***Abbildung 1.8** POS Entitäten*

A. Pakete

Die UML Pakete [29, 30] gruppieren Elemente aus dem Modell und bieten ein Namespace für die gruppierten Elemente an. Die Pakete dürfen Datentypen und andere Pakete enthalten. Ein Datentyp oder ein Paket kann zu einem einzigen Paket gehören.

Bezüglich einer Programmiersprache, entsprechen UML Pakete Java Paketen oder C++/C# namespaces. Die UML Pakete werden mit dem Resolution Operator :: verwiesen, sowohl in C++ als auch in C#. Zum Beispiel, der komplette Name des Paketes *UI* aus den Abbildungen 1.6 oder 1.7 ist *pos::ui.*

Die Klassendiagramme, welche die Pakete eines Systems darstellen, werden dafür benutzt, die Architektur des Systems zu beschreiben – siehe Abbildung 1.6 für das *POS*

System. Die Abhängigkeiten zwischen den Paketen zeigen einen Überblick der Abhängigkeiten zwischen den enthaltenen Elementen und den Elementen aus anderen Paketen. Aus der Perspektive der Architektur, ist eine gute Verwaltung dieser Abhängigkeiten für das Entwicklungs- und Erhaltungsprozess sehr wichtig.

B. Klassen

Eine UML Klasse [29, 30] stellt eine Menge von Objekten mit denselben strukturellen Elemente (Eigenschaften) und Verhaltenselemente (Operationen) dar. Die UML Klassen sind Datentypen und entsprechen den Anwendungsklassen in den Java, C++ und C# Sprachen. Eine Klasse darf als **abstrakt** definiert werden. In diesem Fall darf diese Klasse in Java, C++ und C# nicht instanziiert werden.

Eine UML Klasse kann aus mehreren anderen Klassen **abgeleitet** werden. Die Benutzung mehreren Vererbungen in dem Modell hat nicht unbedingt eine direkte Verbindung mit dem Code in Java, C++ und C#.

Eine UML Klasse kann mehrere Interfaces **implementieren** sowie in Java oder C#. Die Verbindung zwischen den Modellen welche solche Klassen enthalten, die mehrere Interfaces implementieren, und C++ wird durch reine abstrakt C++ Klassen gemacht.

Alle Klassen aus der Abbildung 1.8 sind konkrete Klassen, und *AbstractSaleRepository* aus der Abbildung 1.7 ist eine abstrakte Klasse.

Das **Subtyp-Vererbung Prinzip** gilt für alle Instanzen die eine Klasse oder Interface als Typ haben, sowie in Java, C++ und C#. Die erbende Klasse ist ein Subtyp der Basisklasse im Sinne eines abstrakten Datentyps. Das Prinzip bedeutet, dass ein Objekt des Subtyps (eine Instanz der abgeleiteten Klassen) an jeder Stelle eingesetzt werden kann, an der ein Objekt des Basistyps erwartet wird, ohne die Anwendung semantisch zu ändern.

C. Interfaces

Ein UML Interface (Schnittstelle) [29, 30] ist ein Datentyp, der eine Menge von Operationen definiert, d.h. ein Vertrag (a contract). D.h. ein Interface enthält kein Verhalten, sondern nur die Spezifikation von Methoden und nicht deren Implementierung. Eine Klasse, die ein Interface implementiert muss die definierten Operationen enthalten (den Vertrag erfüllen). Dieses Konzept entspricht demselben Konzept aus Java/C# und den rein abstrakten Klassen in C++.

SaleRepository aus der Abbildung 1.7 ist ein Interface. Wenn das Hervorheben der Methoden des Interfaces nicht relevant ist, dann ist die graphische Darstellung für Interfaces diejenige aus Abbildung 1.9.

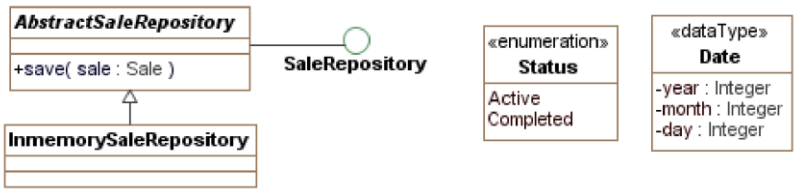


Abbildung 1.9 Interface, Aufzählung und strukturierte Typen

D. Aufzählungen und Wert Objekte

UML Aufzählungen [29, 30] beschreiben eine Menge von Symbolen, die keine assoziierten Werte haben, ähnlich wie in Java, C++ und C#.

Die **strukturierten Typen** [29, 30] werden mit dem *datatype* Stereotyp modelliert und

entsprechen dem C++/C# Strukturen und dem primitiven Typen aus Java. Die Instanzen dieser Datentypen werden durch ihre Werte identifiziert. Diese werden auch dafür benutzt, die Klassen Eigenschaften zu beschreiben, und sie entsprechen den Werte Objekten (the value object[10] pattern), aber sie können keine Identität haben.

E. Verallgemeinerung und Interface Realisierungsabhängigkeit

Die Verallgemeinerung [29, 30] ist eine Beziehung zwischen einem Basistyp (generellen Typ) und einem abgeleiteten Typ (spezialisierten Typ). Diese Beziehung kann zwischen zwei Klassen oder zwischen zwei Interfaces stattfinden, entsprechend der Vererbung Beziehungen zwischen Klassen aus Java und C++/C#, beziehungsweise Interfaces (rein abstrakte Klassen in C++).

Die Realisierung eines Interfaces (Interface-Realisierungsabhängigkeit**)** in UML [29, 30] stellt eine Beziehung zwischen einer Klasse und einem Interface dar und weist darauf hin, dass die Klasse den Vertrag definiert in dem Interface erfüllt. Diese Realisierungsbeziehungen entsprechen den Interfaces Implementierungen in Java und C#, beziehungsweise der Vererbung in C++. Siehe die graphischen Notationen zwischen *AbstractSaleRepository* und *SaleRepository* in **Abbildung 1.7** und **Abbildung 1.9**.

F. Eigenschaften

Eigenschaften [29,30] stellten strukturellen Aspekte eines Datentyps dar. Die Eigenschaften einer Klasse werden durch Attributen und Assoziationen eingeführt. Ein **Attribut** beschreibt eine Eigenschaft einer Klasse (in dem zweiten Teil der Klasse), wie folgt:

visibility name: *type multiplicity* = *value* {*properties*}

Der **Name** ist verpflichtend, sowie die **Sichtbarkeit (visibility)**, die folgende Werte haben kann: + (public), # (protected), - (private) oder ~ (package, wenn kein Symbol da steht, dann ist package default). Die Syntax und Bedeutung der verschiedenen Werte für Sichtbarkeit entspricht in etwa den Sichtbarkeiten in Java. Die Sichtbarkeit auf der Ebene der Pakete ist in C++ nicht vorhanden. In C# gibt es eine Verbindung durch das *internal* Schlüsselwort, der aber auch andere Bedeutungen in der Distribution der Software Elementen hat (auf Elemente, die über den Zugriffsmodifizierer internal verfügen, kann nur in binären dll oder exe Distributionen zugegriffen werden, welche diese Elemente enthalten).

Die anderen Elemente, die benutzt werden um eine Eigenschaft zu definieren, sind optional. **Der Typ** einer Eigenschaft kann folgende Werte haben: Klasse, Interface, Aufzählung, strukturierter Typ oder primitiver Typ (Basisdatentyp). **Basisdatentypen in UML** sind Werttypen [29]. UML definiert folgenden Werttypen: String, Integer, Boolean und UnlimitedNatural. Die ersten drei Basisdatentypen entsprechen den Datentypen in Java, C++ und C#, aber:

- Der String Datentyp ist eine Klasse in Java und C# und die Instanzen eines String Typs können nicht geändert werden, im Vergleich zu C++ wo Strings geändert werden können. Die Zeichenkodierung wird in UML nicht angegeben, während in Java und C# Unicode benutzt wird, und in C++ ASCII.
- Der Integer Datentyp in UML hat unbegrenzte Präzision, während in Java, C++ und C# eine begrenzte Menge von Werten vorhanden ist.

Die **Multiplizität** darf die Werte *1* (default Wert, wenn keine Multiplizität angegeben wird), *0..1* (optional), *0..** (null oder mehrere Werte), *1..** (einer oder mehrere Werte), *m..n* (zwischen *m* und *n* Werte, wobei *m* und *n* Konstanten sind, *n* kann auch * sein) nehmen. Für eine Eigenschaft mit der Multiplizität ***m.. **** können wir zusätzlich Folgendes angeben:

[10] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

- Die Werte können wiederholt werden oder nicht – **implizit sind die Werte eindeutig (unique),** also eine Menge; ansonsten muss explizit spezifiziert werden, dass der Behälter auch Duplikate enthalten kann, mit dem Schlüsselwort **non-unique.**
- Die Werte können durch Indizes bezeichnet werden oder auch nicht – **implizit können diese nicht durch Indizes bezeichnet werden** (d.h. Collection) ; ansonsten muss explizit das Schlüsselwort **ordered** benutzt werden (d.h. Liste).

Beispiele von Eigenschaften:

set *: Integer[0..*]* – Menge von Integer Werten (unique)

list *: Integer[0..*] {ordered}* – Liste mit deutlichen (distinct) Integer Werten (unique)

list *: Integer[0..*] {ordered, non-unique}* – Liste von Integer Werten

collection *: Integer[0..*] {non-unique}* – Collection von Integers

Die UML Eigenschaften entsprechen den Felder oder Objektypen aus Java und C++, beziehungsweise den Eigenschaften in C#. Interpretationsprobleme können vorkommen, wenn die Eigenschaften eine *m.. ** Multiplizität haben. Für die obigen Beispiele nehmen wir folgende Java Korrespondenzen (und ähnlich für C++ und C#):

- Menge von Integers:
  1. *int[] set* oder *Integer[] set,* und wir stellen sicher durch Operationen, dass *set* keine Duplikate enthält, oder angemessener:
  2. *java.util.Set set*
- Liste mit Integers und deutliche Werte:
  1. *int[] list, Integer[] list* oder *java.util.List list,* und wir stellen sicher durch Operationen, dass *liste* keine Duplikate enthält
- Liste von Integers:
  1. *int[] list, Integer[] list,* oder *java.util.List list*
- Collection von Integers:
  1. *int[] collection, Integer[] collection,* oder *java.util.Collection collection*

UML Assoziationen [29, 30] sind eine Menge von Tupeln, wobei jeder Tupel zwei Instanzen von irgend zwei Datentypen miteinander verbinden. Eine Assoziation ist ein Datentyp, der die Eigenschaften zweier Datentypen miteinander verbindet.

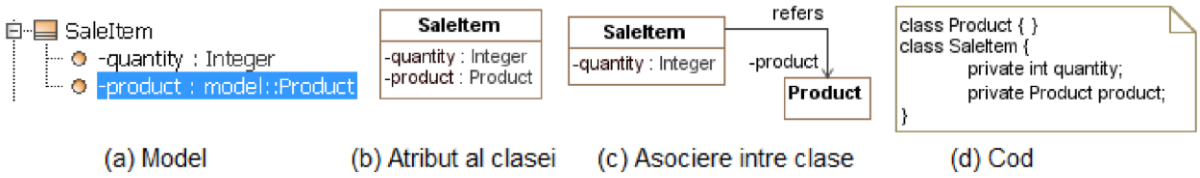


Abbildung 1.10 Unidirektionale Assoziationen

Abbildung 1.10 (a) stellt das Modell dar, das nach dem Einfügen der Attributen *quantity* und *product* zu der Klasse *SaleItem* aus der Diagramm (b), resultiert. Der Code (d) geschrieben in Java/C# entspricht dieser Situation. Wenn wir eine graphische Darstellung der Beziehung zwischen den Klassen *SaleItem* und *Product* als passender halten, dann können wir, anstatt *product* als Attribut hinzufügen, eine unidirektionale Assoziationbeziehung von *SaleItem* nach *Product* definieren. Wenn eine unidirektionale Assoziationsbeziehung eingefügt wird, dann wird eine Assoziation in dem Modell erstellt und eine Eigenschaft in der *SaleItem* Klasse eingefügt, mit demselben Rollennamen *product.* Auf dieser Weise entspricht der Code (d) der graphischen Darstellung (c) des Modells (a), der auch eine Assoziation enthält, die nicht in der Abbildung dargestellt wird. Die

unidirektionale Assoziation fügt Eigenschaften in der Source Klasse ein, mit dem Typ der Zielklasse. Der Name der Eigenschaft ist gleich mit dem Namen der Assoziationsrolle, und die allgemeine Form für das Definieren der Eigenschaften (beschrieben am Anfang dieses Abschnittes) ist in diesem Fall auch anwendbar.

Die Entscheidung Assoziationen anstatt Eigenschaften zu benutzen hängt von dem Kontext ab. Zum Beispiel, wenn wir Entitäten aus einer Anwendung modellieren, dann benutzen wir Assoziationen um Beziehungen zwischen den Entitäten zu zeigen, und Attributen um die Entitäten mithilfe von beschreibende Wertobjekte zu beschreiben. Meistens werden Assoziationen dafür benutzt, die Bedeutung der Typen und der Beziehungen dazwischen hervorzuheben.

Bidirektionale Assoziationen verbinden zwei Eigenschaften aus unterschiedlichen Klassen oder aus derselben Klasse. **Abbildung 1.11** zeigt eine bidirektionale Assoziation zwischen *SaleItem* und *Product*, sowie der entsprechende Java/ C# Code für diese Situation.

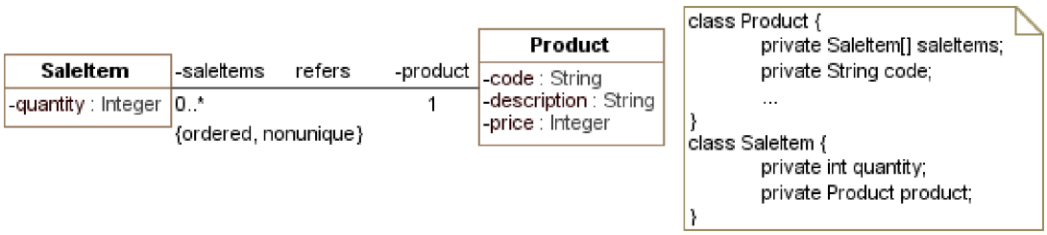


Abbildung 1.11 Bidirektionale Assoziationen

Der konzeptuellen Modelle enthalten bidirektionale Assoziationen. Das Speichern der bidirektionalen Assoziationen in den PIM/PSM Modellen kann zu einer ineffizienter Ausführung (execution) führen wegen der Objektrepresentierung. Ein verpflichtender Schritt für die detaillierte Analyse ist die **Verfeinerung der Assoziationen**, wobei erstmal **bidirektionale Assoziationen in unidirektionale Assoziationen umgewandelt werden**. **Abbildung 1.8** zeigt das Ergebnis der Verfeinerung der Assoziationen aus **Abbildung 1.5**.

Die Teil-Ganzes-Beziehungen werden in UML als Aggregationen und Kompositionen modelliert. Eine **Aggregation** wird auch als Ist-Teil-von-Beziehung bezeichnet und gibt an, dass eine Klasse in einer anderen Klasse „enthalten“ ist. Zum Beispiel, in Abbildung 1.7, die Aggregation zwischen *InmemorySaleRepository* und *Sale* gibt an, dass das erste Objekt alle Objekte vom Typ *Sale* speichert (*Sales* sind Teil dieses Repository). Eine **Komposition** ist eine stärkere Form der Aggregation, die angibt, dass eine Klasse aus anderen „besteht“, d.h. zusätzlich kann das enthaltene Objekt nur zu einer Klasse gehören. Zum Beispiel, in Abbildung 1.8, ist ein Element des Verkaufs (*SaleItem*) Teil eines einzigen Verkaufs (*Sale*). Die **Verfeinerung der Assoziationen** umfasst auch die Festlegung der **Aggregation** und **Komposition** Beziehungen.

Sowie auch in Java, C++ und C#, kann man **statische oder Klassen Typen Eigenschaften** definieren. In den Diagrammen werden diese als unterstrichen dargestellt.

G. Abhängigkeiten

Zwischen zwei Software Elemente, *Kunde* und *Lieferant*, gibt es eine **Abhängigkeit** [29, 30] wenn eine Änderung der Definition des *Lieferanten* zu der Änderung des *Kunden* führt. Zum Beispiel, wenn eine Klasse C eine Nachricht zu einer anderen Klasse F schickt, dann ist C abhängig von F weil eine Änderung der Definition der Nachricht in F zu Änderungen in C führt bezüglich der Übertragungsart. Als eine allgemeine Regel, sollte man die Abhängigkeiten in dem Modell minimieren, während man die Elemente kohäsiv behält.

Man kann die Abhängigkeiten zwischen jedwelchen Elementes aus dem Modell in UML explizit angeben. Aber mit der expliziten Angabe kann der Modell schwer zu lesen werden. Darum werden Abhängigkeiten selektiv dargestellt, wobei nur wichtige Abhängigkeiten und Elemente der Architektur hervorgehoben werden – siehe Abbildung 1.6 und 1.7.

H. Operationen

Operationen in UML [29, 30] definieren das Verhalten der Objekte und entsprechen den Methoden der Objekte in objektorientierten Programmiersprachen. Eigentlich, wird das Verhalten (stellt die Signatur dar) von den Operationen bestimmt und der Methodentext (body of a method) wird von den Verhaltenselementen, sowie Interaktionen, Zustandsmaschinen (state machines) und Aktivitäten, definiert. Implementierungen werden als **Methoden** in UML bezeichnet. Die Syntax um eine Operation zu spezifizieren ist:

visibility name (*list-of-parameters*) : *returned-type* {*properties*}

wobei *visibility, returned-type* und *properties* sowie die Klassen-Eigenschaften definiert sind. In der Liste der Operationen von *properties* kann man angeben ob es nur eine Query Operation ist {*query*}, d.h. eine Operation welche den Zustand des Objektes nicht ändert – implizit, gelten Operationen als Befehle, d.h. Operationen welche den Zustand der Objekte ändern. Die Parameter aus der *list-of-parameters* werden durch Kommas getrennt, wobei ein Parameter folgende Form hat:

direction name: *type* = *implicit-value*,

direction kann folgende Werte nehmen: *in, out* und *in-out* (default: *in*).

1.4.2 Interaktionsdiagramme

Die UML Interaktionen [29, 30] beschreiben das Verhalten des Systems, indem sie angeben wie mehrere Teilnehmer die in einem Szenario beteiligt sind zusammenarbeiten. Es gibt im Wesentlichen drei Hauptvarianten von Interaktionsdiagrammen: Sequenzdiagramme, Zeitdiagramme und Kommunikationsdiagramme. In diesem Abschnitt besprechen wir das **Sequenzdiagramm**, das den Nachrichtenaustausch zwischen mehreren Interaktionspartnern darstellt.

Sequenzdiagramme sind besonders dann gut geeignet, wenn eine Interaktion wenige Interaktionspartner hat, die viele Nachrichten austauschen, beziehungsweise deren Interaktionsmuster komplex ist. Meistens beschreibt das Sequenzdiagramm **ein einziges Szenario**. Zum Beispiel, Abbildung 1.12 zeigt den Kassierer und das *POS* System, wobei der normale Ablauf für den einzigen besprochenen Anwendungsfall (Use Case) beschrieben wird, das Speichern eines Verkaufs. Ein solches Diagramm hilft das **öffentliche (public) Interface des Systems** zu bestimmen. Ausgehend von den Anwendungsfällen, werden die Aktionen der Benutzer als Nachrichten modelliert auf welche das System antworten muss. Die Nachrichten 1, 2, 4 und 6 aus **Abbildung 1.12** deuten an, dass das System etwas berechnen muss um dem Benutzer zu antworten.

Das Sequenzdiagramm beschreibt die zeitliche Abfolge von Interaktionen zwischen einer Menge von Objekten innerhalb eines zeitlich begrenzten Kontextes. Die **Zeitlinie** verläuft senkrecht von oben nach unten, die Objekte werden durch senkrechte Lebenslinien beschrieben und die gesendeten Nachrichten waagrecht entsprechend ihres zeitlichen Auftretens eingetragen. Die Objekte werden durch Rechtecke visualisiert. Von Ihnen aus gehen die senkrechten Lebenslinien, dargestellt durch gestrichelte Linien, ab. Die

Nachrichten werden durch waagerechte Pfeile zwischen den Objeklebenslinien beschrieben. Auf diesen Pfeilen werden die Nachrichtennamen in der Form: *nachricht(argumente)* notiert. Nachrichten können nummeriert werden und sie können auch eine **Antwort (response)** angeben. Alle Nachrichten aus den Abbildungen in diesem Abschnitt sind synchron. Objekte, die gerade **aktiv an Interaktionen beteiligt** sind, werden durch einen **Balken auf ihrer Lebenslinie** gekennzeichnet. Die Interaktionen können **Fragmente** enthalten: **Zyklen** und **Alternativen**.

Diagramme ähnlich mit dem Diagramm aus der Abbildung 1.12 können in dem CIM Kontext definiert werden, bevor man die Architektur bestimmt. Auf der Ebene der PIM, nach der Indentifizierung des Verhaltens eines Systems (was das System machen muss), kann man ein Sequenzdiagramm benutzen um zu detaillieren, wie die Objekte aus dem System zusammenarbeiten (collaborate), gemäß der Verantwortungen, die durch die ausgewählte Architektur bestimmt wurden. Abbildung 1.14 zeigt die detaillierte Zusammenarbeit (collaboration) in Falle des *POS* Systems.

Die Teilnehmer aus der **Abbildung 1.12** geben keine Instanzen irgendwelcher Typen aus dem Modell an. Diesmal, die Hauptteilnehmer aus der Abbildung 1.14 sind Objekten mit dem Typ *controller, service* und *repository*, gemäß der *POS* Architektur. Das Diagramm zeigt die Nachrichten adressiert an dem Controller (1, 3, 6 und 9), da in dem Diagramm der User Interface Element nicht vorhanden ist.

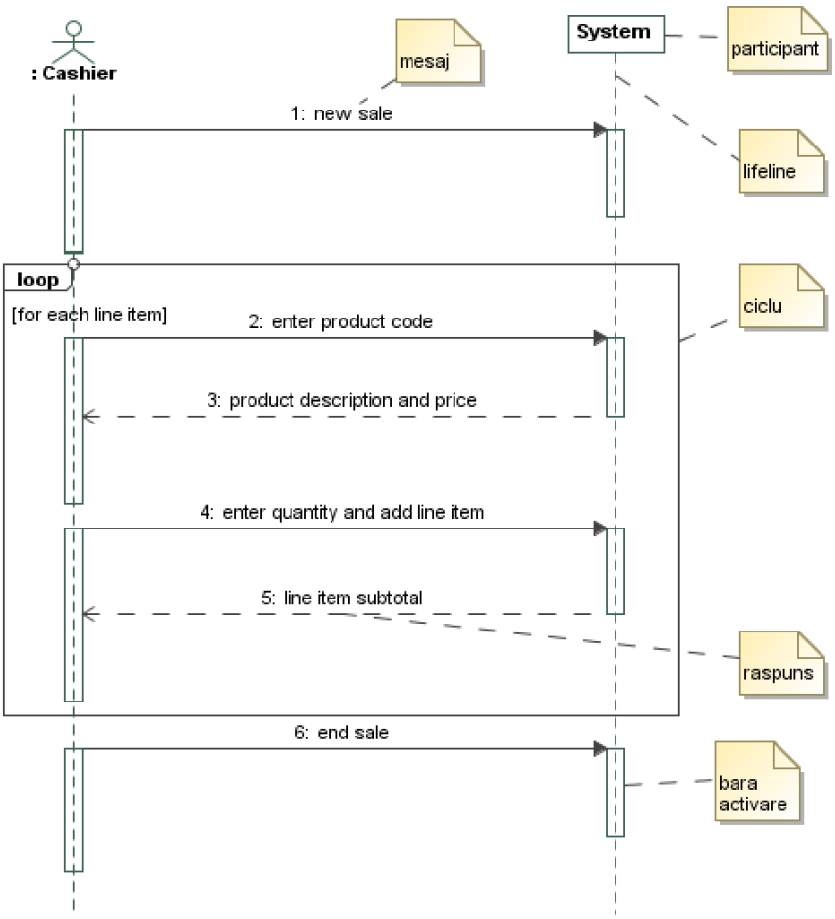


Abbildung 1.12 Die Interface eines Systems

Da die Teilnehmer Objekte darstellen, sind die gesendeten Nachrichten Methodenaufrufe (method call) für die Objekte an denen die Nachrichten geschickt werden. Also, führt das Diagramm zu der Identifizierung der Methoden von Objekten. Abbildung 1.13 stellt die Methoden dar, die basierend auf den Interaktionen aus Abbildung 1.14 identifiziert wurden.

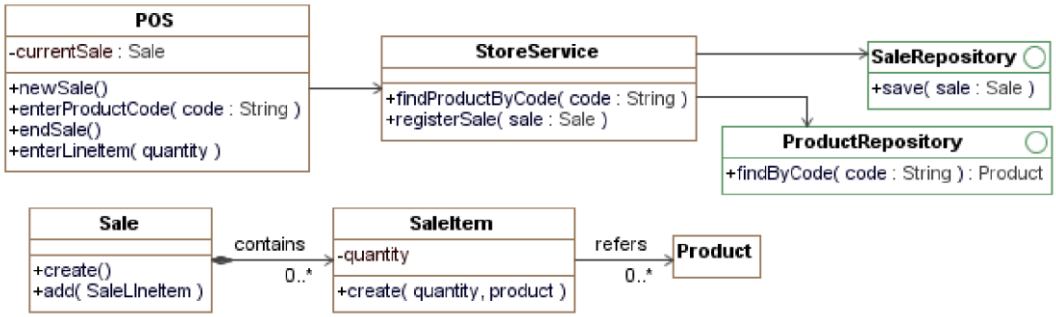


Abbildung 1.13 Detailliertes Design - Klassendiagramm

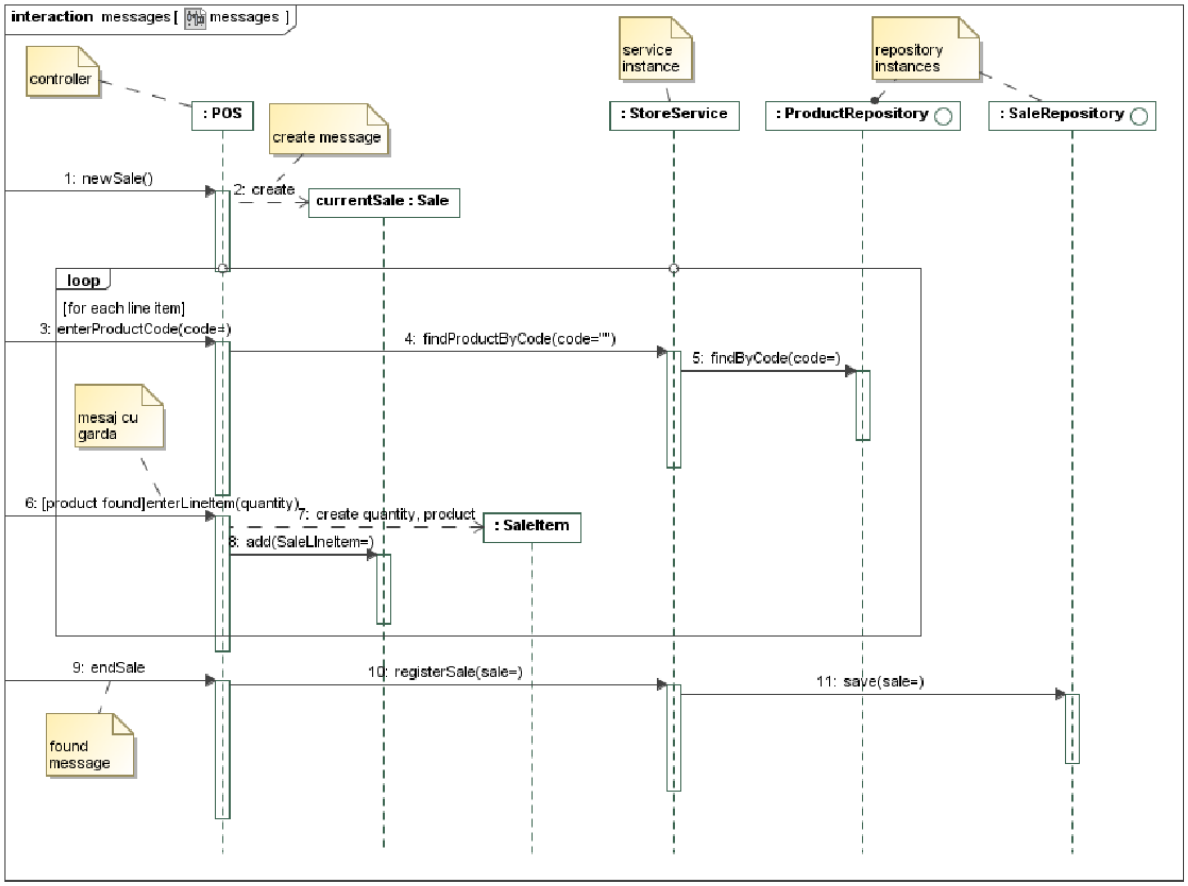


Abbildung 1.14 Detailliertes Design - Interaktionen zwischen Objekten

1.5 Listen und assoziative Datenfelder

Im Folgenden werden zwei häufige Datenstrukturen und deren zulässigen Operationen beschrieben: ***Listen*** und ***assoziative Datenfelder***. Für jede Operation wird in natürlicher Sprache eine kurze Beschreibung und die Nach- und Vorbedingungen abgibt.

1.5.1. Listen

Das Wort *Liste* bezieht sich im Umgangssprache eine Sammlung von Daten (Personen, Studenten, Preise, Einkäufe usw.) in einer festgelegten Reihenfolge. Diese Reihenfolge kann als eine "Verknüpfung" zwischen Elementen interpretiert werden: nach dem ersten Einkauf folgt die zweite, nach dem zweiten folgt die dritte usw. Es gilt auch als ein Rang des Elements in der Liste (der erste Einkauf, der zweite Einkauf usw.). Der `List`-DatenTyp, der im Folgenden definiert wird erlaubt Anwendungen in solchen realen Lebenssituationen.

Eine **Liste** gilt daher als ein Feld von Elemente $<l_1,l_2,...,l_n>$ mit dem gleichen Typ (`TElement`) in einer festgelegten Reihenfolge. Jedes Element hat eine festgelegte *Position* in der Liste. Die Position der Element ist deshalb wichtig. Zugänge, Einfügungen und Löschungen richten sich nach der Position.

Eine **Liste** gilt auch als ein dynamischer Container von Elementen wobei die Reihenfolge der Elemente wichtig ist. Die Anzahl der Elemente *n* ist die *Länge* einer Liste. Eine Liste mit Länge 0 ist eine *leere* Liste. Der dynamische Aspekt der List stammt von ihre dynamische Länge, die durch Einfügungen und Löschungen verändert werden kann.

Im Folgenden werden lineare Liste diskutiert. Die Liste ist eine Datenstruktur, die leer ist oder:

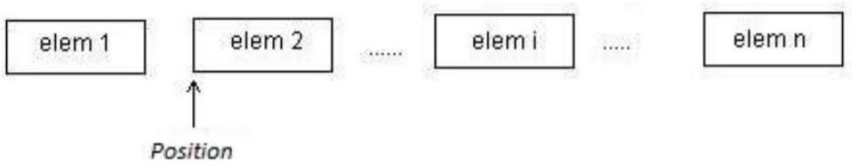
- es enthält ein einzigartiges erstes Element
- es enthält ein einzigartiges letztes Element
- jedes Element (außer das erste Element) hat einen Nachfolger
- jedes Element (außer das letzte Element) hat einen Vorgänger

Das heißt, dass eine List eine festgelegte Reihenfolge hat. Für eine Position kann man das Element der entsprechenden Position und seinen Nachfolger oder Vorgänger (falls möglich) ermitteln.

Wenn das letzte Element auf den Listenkopf zeigt (und umgekehrt), ist eine Liste als **zyklische** genannt.
Die **Position** eines Elements kann in verschiedene Möglichkeiten betrachtet sein:

1. als der **Rang** des Elements in der Liste. Diese Situation ist vergleichbar mit Felder: die Position eines Elements ist seiner Rang in der Liste. In einem solchen Fall, kann die Liste als ein dynamisches Feld interpretiert sein.
2. als ein **Pointer** auf dem Speicherplatz des Elements.

Um die Allgemeingültigkeit zu sichern, kann man das Konzept der ***Position*** generalisieren. Somit können Elemente der Liste mittels dieses Konzept zugegriffen werden.



Eine Position p in der Liste ist **gültig**, wenn es die Position eines Element der Liste ist. Zum Beispiel, falls p ein Pointer auf ein Element der Liste verbunden ist, ist p **gültig** wenn es nicht NULL ist oder wenn es auf dem Speicherplatz der Liste Zeigt. Falls p ein Rang des Elementes ist, ist p **gültig** wenn es nicht größer als die Anzahl der Elemente ist.

Im folgenden wird ein abstrakter Datentyp **List** beschreiben. Die typische Operationen sind: suchen, einfügen und entfernen von Elemente für bestimmte Positionen. Der spezielle Wert ⊥zeigt dass eine Position nicht gültig ist. Der Wert ϕ zeigt eine leere Liste.

`Abstrakter Datentyp LIST`

Domain

DList(Position; **TElement**) = *{l* | *l* ist ein Element vom Typ **TElement**; jedes Element hat eine Position vom Type *Position*}

Im folgenden die Notation **L**= **D**List(Position; `TElement`) wird benutzt.

Operationen

create *(l)*

Beschreibung: erzeugt eine leere Liste
Vorbedingung: true
Nachbedingung: *l*∈**L**, *l* =ϕ

valid*(l,p)*

Beschreibung: prüft ob eine Position gültig ist
Vorbedingung: *l*∈**L,** *p*∈*Position*
Nachbedingung: *valid*=true falls *p* die Position einem Element *e* von *l* ist
valid=false sonst

addLast*(l,e)*

Beschreibung: fügt ein Element am Ende der Liste ein
Vorbedingung: *l*∈**L,** *e*∈`TElement`
Nachbedingung: *l'*∈**L**, *l'* ist die neue Liste

addFirst*(l,e)*

Beschreibung: fügt ein Element am Anfang der Liste ein
Vorbedingung: *l*∈**L,** *e*∈`TElement`
Nachbedingung: *l'*∈**L**, *l'* ist die neue Liste

addBefore*(l,p,e)*

Beschreibung: fügt ein Element vor einer Position ein
Vorbedingung: *l*∈**L**, *e*∈TElement, *p*∈*Position,* ***valid****(l,p)*
Nachbedingung: *l'*∈**L**, *l'* ist die neue Liste

addAfter*(l,p,e)*

Beschreibung: fügt ein Element nach einer Position ein
Vorbedingung: *l*∈**L**, *e*∈TElement, *p*∈*Position,* ***valid****(l,p)*
Nachbedingung: *l'*∈**L**, *l'* ist die neue Liste

delete*(l,p,e)*

Beschreibung: entfernt ein Element der Liste
Vorbedingung: *l*∈**L**, *e*∈TElement, *p*∈*Position,* ***valid****(l,p)*
Nachbedingung: *e*∈TElement, *l'*∈**L**, *l'* ist die neue Liste ohne *e*, *e* ist das entfernte Element

getElement*(l,p,e)*

Beschreibung: gibt das Element *e* an Position *p* zurück
Vorbedingung: *l*∈**L**, *e*∈TElement, *p*∈*Position,* ***valid****(l,p)*
Nachbedingung: *e*∈TElement, *e* ist das Element an Position *p*

setElement*(l,p,e)*

Beschreibung: ändert das Element *e* an Position *p*
Vorbedingung: *l*∈**L**, *e*∈TElement, *p*∈*Position,* ***valid****(l,p)*
Nachbedingung: *l*∈**L**, *l* ist die neue Liste

first*(l)*

Beschreibung: gibt das erste Element der Liste zurück
Vorbedingung: *l*∈**L**
Nachbedingung: *first*∈*Position*
first=⊥ falls die Liste leer ist

last*(l)*

Beschreibung: gibt das letzte Element der Liste zurück
Vorbedingung: *l*∈**L**
Nachbedingung: *last*∈*Position*
last=⊥ falls die Liste leer ist

next*(l,p)*

Beschreibung: gibt die nächste Position *nach p* zurück
Vorbedingung: *l*∈**L**, *p*∈*Position,* ***valid****(l,p)*
Nachbedingung: *next*∈*Position*
next=⊥ falls *p* die letzte Position der Liste ist

previous*(l,p)*

Beschreibung: gibt die nächste Position *vor p* zurück
Vorbedingung: *l*∈**L**, *p*∈*Position,* ***valid****(l,p)*

Nachbedingung: *previous*∈*Position*
previous=⊥ falls *p* die erste Position der Liste ist

search*(l,e)*
Beschreibung: sucht ein Element der Liste
Vorbedingung: *l*∈**L**, *e*∈TElement
Nachbedingung: *search*∈*Position*
search=⊥ falls *e*∉ *l*

isIn*(l,e)*
Beschreibung: prüft ob ein Element in der Liste ist
Vorbedingung: *l*∈**L**, *e*∈TElement
Nachbedingung: *isIn*= true falls *e*∈ *l*
isIn=false sonst

isEmpty*(l)*
Beschreibung: prüft ob die Liste leer ist
Vorbedingung: *l*∈**L**
Nachbedingung: *isEmpty*= true falls *l*=*ϕ*
isEmpty=false sonst

size*(l)*
Beschreibung: gibt die Anzahl der Elemente zurück
Vorbedingung: *l*∈**L**
Nachbedingung: *size*=die Anzahl der Elemente

getIterator*(l)*
Beschreibung: erzeugt ein Iterator der Liste
Vorbedingung: *l*∈**L**
Nachbedingung: *i*=ein Iterator der Liste

destroy*(l)*
Beschreibung: entfernt die Liste
Vorbedingung: *l*∈**L**
Nachbedingung: *die liste wird entfernt*

Man kann eine Liste durch den Iterator ausgeben.

```
Subalgorithm print (l):
{pre: l is a list }
{post: the elements of the list are printed }

      getIterator(l, i)              {get the iterator on the list l}
      While valid(i) do                 {while the iterator is valid}
            currentElement(i, e)            {e is the current element}
            @ print e                      {print the current element}
            next(i)              {iterator refers to the next element}
      endWhile
```

`end-print`

Anmerkung

Es gibt keine allgemeingültige Normen für Spezifikationen der Operationen. Andere äquivalente Spezifikationen für die *addLast* Operation:

***addLast**(l,e)*

Beschreibung: fügt ein Element am Ende der Liste ein
Vorbedingung: $l \in \mathbf{L}$, $e \in$ `TElement`
Nachbedingung: $l' \in \mathbf{L}$, $l' = l \cup \{e\}$, e ist das letzte Element von l'

***addLast**(l,e)*

Beschreibung: fügt ein Element am Ende der Liste ein
Vorbedingung: $l \in \mathbf{L}$, $e \in$ `TElement`
Nachbedingung: $l \in \mathbf{L}$, e wird am ende der Listen eingefügt; die andere Positionen werden nicht verändert

1.5.2. Assoziative Datenfelder

Das assoziative Datenfelder ist eine Datenstruktur, die Paare von Schlüssel und Werte (key, value) enthält. Assoziative Datenfelder verwenden Schlüssel, um die enthaltenen Elemente einfach zu adressieren. Das heißt, dass die Schlüssel einzigartig sind und jede Schlüssel hat einen assoziiert Wert. Die typische Operationen sind: suchen, einfügen und entfernen von Elemente.

Assoziative Datenfelder haben viele Anwendungen. Um nur einige zu nennen:

- Informationen über Bankkonten: Jedes Konto kann durch eine Kontonummer (der Schlüssel) und weitere Informationen wie z.B. Kontoinhaber, Kontostand (der Wert) identifiziert sein.
- Informationen über Telefondienstkunden: Jeder Kunde kann durch eine Telefonnummer (der Schlüssel des Paars) und weitere Informationen wie z.B. Name, Adresse (der Wert) identifiziert sein.
- Informationen über Studenten: Jeder Student kann durch eine Matrikelnummer (der Schlüssel) und weitere Informationen wie z.B. Name, Adresse (der Wert) identifiziert sein.

Im folgenden wird ein abstrakter Datentyp **Map** beschreiben. **TKey** beschreibt den Typ des ersten Element eines Paars (der Schlüssel). **TValue** beschreibt den Typ des zweiten Element eines Paars (der Wert). Der spezielle Wert $\perp$ist identisch zu NULL/Nil in Programmiersprachen.

`Abstrakter Datentyp MAP`

Domain

$\mathbf{D}$Map(TKey, TValue) = {**m** | **m** ist ein Gebinde von **(k,v)**, wobei k: TKey, v:TValue}

Im folgenden die Notation $\mathbf{D}$ = $\mathbf{D}$Map(TKey, TValue) wird benutzt.

Operationen

create *(d)*

Beschreibung: erzeugt ein leeres assoziatives Datenfeld
Vorbedingung: true
Nachbedingung: *d*∈$\mathbf{D}$, *d*=ϕ

add*(d,c,v)*

Beschreibung: fügt ein Paar ein
Vorbedingung:*d*∈$\mathbf{D}$, ***c***∈**TKey**, ***v***∈**TValue**
Nachbedingung: *d'*∈$\mathbf{D}$, *d'*=$d \cup \{(c, v)\}$

search*(d,c)*

Beschreibung: sucht ein Element des assoziativen Datenfeldes
Vorbedingung: *d*∈$\mathbf{D}$, ***c***∈**TKey**
Nachbedingung: *search*∈**TValue**
search=$\boldsymbol{v}$ falls (c,v)∈ d
search=⊥ sonst

remove*(d,c)*

Beschreibung: entfernt ein Element des assoziativen Datenfeldes
Vorbedingung: *d*∈$\mathbf{D}$, ***c***∈**TKey**
Nachbedingung: *remove*∈**TValue**
remove=$\boldsymbol{v}$ falls (c,v)∈ d
remove=⊥sonst
d'=d ohne (c,v) falls (c,v)∈ d
d'=d sonst

isEmpty*(d)*

Beschreibung: prüft ob das assoziative Datenfeld leer ist
Vorbedingung: *d*∈$\mathbf{D}$
Nachbedingung: *isEmpty*= true falls *d*=ϕ
isEmpty=false sonst

size*(d)*

Beschreibung: gibt die Anzahl der Elemente zurück
Vorbedingung: *d*∈$\mathbf{D}$
Nachbedingung: *size*=die Anzahl der Elemente

keySet*(d,m)*

Beschreibung: gibt die Menge der Schlüssel des assoziativen Datenfeldes zurück
Vorbedingung: *d*∈$\mathbf{D}$
Nachbedingung: *m*∈$\mathbf{M}$, *m* ist die Menge der Schlüssel

***values**(d,c)*

Beschreibung: gibt die Collection der Werte des assoziativen Datenfeldes zurück
Vorbedingung: *d*∈**D**
Nachbedingung: *c*∈**C,** *c* ist die Collection der Werte

***pairs**(d,m)*

Beschreibung: gibt die Menge von (key, value) zurück
Vorbedingung: *d*∈**D**
Nachbedingung: *m*∈**M,** *m* ist die Menge von (key, value)

***getIterator**(d)*

Beschreibung: erzeugt ein Iterator des assoziativen Datenfeldes
Vorbedingung: *d*∈**D**
Nachbedingung: *i*=ein Iterator des assoziativen Datenfeldes

***destroy**(d)*

Beschreibung: entfernt das assoziative Datenfeld
Vorbedingung: *d*∈**D**
Nachbedingung: *das Feld wird entfernt*

Man kann ein assoziatives Datenfeld durch den Iterator ausgeben.

```
Subalgorithm print (m):
{pre: m is a map}
{post: the elements of the map are printed }

     getIterator(l, i)              {get the iterator on the map m}
     While valid(i) do                 {while the iterator is valid}
          currentElement(i, e)            {e is the current element}
          @ print e                      {print the current element}
          next(i)              {iterator refers to the next element}
     endWhile
end-print
```

1.6 Aufgaben

I. Schreiben Sie ein Python, Java, C++ oder C# Programm.

a. Implementieren Sie eine Klasse **B** mit einem Integer **b** und einer Methode, die das Attribut **b** auf dem Bildschirm ausgibt.
b. Implementieren Sie eine Klasse **D** mit einem String **d**, die von **B** erbt. Klasse **D** hat auch eine Methode, die die Attributen **b** und **d** auf dem Bildschirm ausgibt.
c. Schreiben Sie eine Funktion, die eine Liste erstellt. Die Liste enthält: ein Objekt $\mathbf{o_1}$ vom Typ **B** mit **b=8**; ein Object $\mathbf{o_2}$ vom Typ **D** mit **b=5** und **d="D5"**; ein Objekt $\mathbf{o_3}$ vom Typ **B** mit **b=-3**; ein Objekt $\mathbf{o_4}$ vom Typ **D** mit **b=9** und **d="D9"**.

d. Schreiben Sie eine Funktion, die eine Liste von Objekte des Typs **B** bekommt. Die Funktion soll nur Objekten, die die Bedingung **b>6** erfüllt, liefern.
e. Schreiben Sie eine Spezifikation für den benutzten **List-Typ**.

Sie dürfen bestehende Bibliotheken für die Datenstrukturen benutzen (Python, Java, C++, C#). Sie dürfen Listenoperationen nicht implementieren.

II. Schreiben Sie ein Python, Java, C++ oder C# Programm.

a. Implementieren Sie eine Klasse **B** mit einem Integer **b** und einer Methode, die das Attribut **b** auf dem Bildschirm ausgibt.
b. Implementieren Sie eine Klasse **D** mit einem String **d**, die von **B** erbt. Klasse **D** hat auch eine Methode, die die Attributen **b** und **d** auf dem Bildschirm ausgibt.
c. Schreiben Sie eine Funktion, die ein assoziatives Datenfeld erstellt. Das Feld enthält: ein Objekt $\mathbf{o_1}$ vom Typ **B** mit **b=8**; ein Object $\mathbf{o_2}$ vom Typ **D** mit **b=5** und **d="D5"**; ein Objekt $\mathbf{o_3}$ vom Typ **B** mit **b=-3**; ein Objekt $\mathbf{o_4}$ vom Typ **D** mit **b=9** und **d="D9"**. Die Schüssel ist die Wert des Attributes **b.**
d. Schreiben Sie eine Funktion, die ein ein assoziatives Datenfeld von Objekte des Typs **B** bekommt. Die Funktion soll nur Objekten, die die Bedingung **b>6** erfüllt, liefern.
e. Schreiben Sie eine Spezifikation für den benutzten **Feld-Typ**.

Sie dürfen bestehende Bibliotheken für die Datenstrukturen benutzen (Python, Java, C++, C#). Sie dürfen assoziative Datenfeldeoperationen nicht implementieren.

III. Ein **Klassendiagramm** und ein **Interaktionsdiagramm** werden beschrieben werden. Sie müssen **ein Programm,** das die Diagramme erfüllt, **schreiben**.

Sie dürfen das Programm in einer objektorientierten Sprache schreiben (Python, Java, C++, C#).

Teil 2. Datenbanken

2.1. Relationale Datenbanken. Die ersten drei Normalformen einer Relation

2.1.1. Relationales Modell

Das relationale Modell einer Datenbank wurde in 1970 von E. F. Codd eingeführt und ist das am meisten benutzte Datenmodell. Im folgenden Abschnitt wir das relationale Modell eingeführt.

Gegeben sei A_1, A_2, …, A_n eine Menge von Attributen (oder Datenfelder) und D_i = Dom (A_i) ∪ {?} der *Wertebereich* (oder *Domäne*) für das Attribut A_i, wobei „?“ einen „unbekannten“ Wert (Null) darstellt. Man benutzt den Wert „unbekannt“ wenn ein Attribut kein anderes zugewiesenes Wert hat. Das „unbekannte“ Wert gehört nicht zu einem Datentyp und es kann zusammen mit anderen Attributwerte von unterschiedlichen Datentypen (Nummer, Text, Datum, usw.) in Ausdrücken benutzt werden.

Eine Relation R von Grad n (Stelligkeit, degree/arity) ist definiert als eine Teilmenge des kartesischen Produkts der n Domänen:

$$R \subseteq D_1 \times \ldots \times D_n$$

Die Relation R kann als seine Menge von Vektoren mit n Werten betrachten werden, ein Wert für jedes Attribut A_i. So eine Relation kann man folgendermaßen in einer Tabelle repräsentieren:

R	A_1	…	A_j	…	A_n
r_1	a_{11}	…	a_{1j}	…	a_{1n}
…	…	…	…	…	…
r_i	a_{i1}	…	a_{ij}	…	a_{in}
…	…	…	…	…	…
r_m	a_{m1}	…	a_{mj}	…	a_{mn}

wobei die Zeilen Elemente der Relation/Tupeln/Datensätze darstellen, die distinkt sind, und $a_{ij} \in D_j, j = 1, \ldots, n, i = 1, \ldots, m$. Weil die Darstellung der Elemente einer Relation R einer Tabelle ähnelt, wird die Relation Tabelle genannt. Um den Namen und die Attribute der Relation hervorzuheben wird folgende Notation benutzt:

$$R[A_1, A_2, \ldots, A_n].$$

Eine Ausprägung (Instanz) der Relation ([R]) ist eine Teilmenge des kartesischen Produkts $D_1 \times \ldots \times D_n$.

Eine relationale Datenbank ist eine Menge von Relationen. Ein Datenbankschema ist die Menge der Relationenschemata. Eine Datenbank Ausprägung (state) ist die Menge der Ausprägungen aller Relationen aus der Datenbank.

Eine relationale Datenbank enthält drei Teile:

1. **Daten (Inhalt der Datenbank) und Beschreibung der Datenbank**
2. **Integritätsregeln (integrity constraints) welche die Konsistenz der Datenbank versichern**
3. **Operatoren für die Verwaltung der Daten**

Beispiel 1. STUDENTEN [NAME, GEBURTSJAHR, STUDIUM_JAHR],
mit folgenden möglichen Werten:

NAME	GEBURTSJAHR	STUDIUM_JAHR
Surdu Ioan	**1985**	**2**
Barbu Ana	**1987**	**1**
Dan Radu	**1986**	**3**

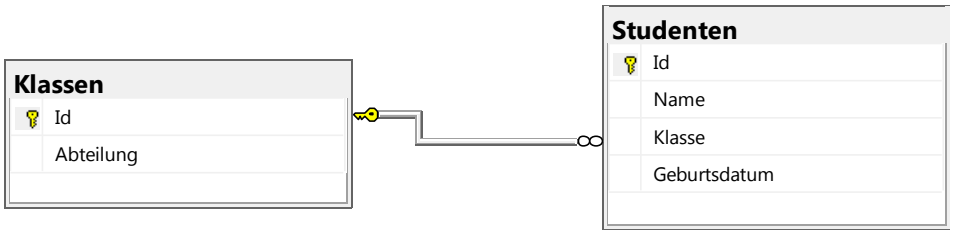
Beispiel 2. BUCH [AUTHOR, TITEL, VERLAG, JAHR]
mit folgenden Werten:

AUTHOR	TITEL	VERLAG	JAHR
Date, C.J.	**An Introduction to Database Systems**	Addison-Wesley Publishing Comp.	2004
Ullman J., Widom, J.	**A First Course in** Database Systems	Addison-Wesley + Prentice-Hall	2011
Helman, P.	The Science of Database Management	Irwin, SUA	1994
Ramakrishnan, R.	Database Management Systems	McGraw-Hill	2007

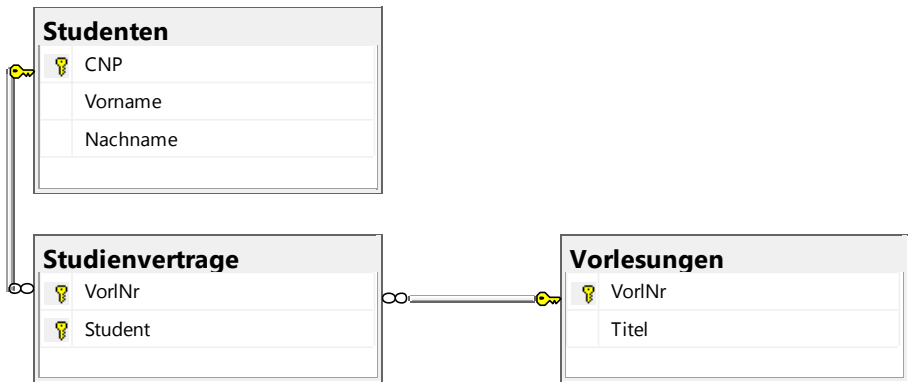
Ein Attribut oder eine Menge von Attributen wird als Schlüsselkandidat/Schlüssel bezeichnet, wenn es keine zwei Tupeln gibt, die genau dieselben Werte haben für alle Attribute aus dieser Menge.
Intuitiv sind Schlüsselkandidate minimale Mengen von Attributen, deren Werte ein Tupel eindeutig identifizieren. Weil alle Tupeln der Relation unterschiedlich sind, gibt es immer wenigstens einen Schlüsselkandidaten (im schlimmsten Fall besteht dieser aus allen Attributen der Relation). In dem 1. Beispiel kann {NAME} als Schlüsselkandidat betrachtet werden (wenn es nicht möglich ist, dass zwei Studenten denselben Namen haben). In dem 2. Beispiel kann die Menge {AUTHOR, TITEL} als Schlüsselkandidat ausgewählt werden und es scheint sogar die einzige Möglichkeit zu sein (oder man fügt ein neues Attribut ISBN ein, das ein Buch eindeutig identifiziert).
Im Allgemeinen kann eine Relation mehrere Schlüsselkandidate besitzen. Einer davon wird als Primärschlüssel ausgewählt, während die anderen als Sekundärschlüssel (oder alternative Schlüssel) betrachten werden. Relationale Datenbank Management Systeme (DBMS) erlauben nicht, dass zwei Tupeln denselben Wert für einen Schlüssel haben. Ein Schlüssel ist eine Integritätsregel für eine Datenbank.
Beispiel 3. STUNDENPLAN [TAG, STUNDE, RAUM, PROFESSOR, GRUPPE, VORLESUNG]
Die Relation STUNDENPLAN speichert die wöchentliche Stundepläne der Fakultät. Es gibt drei Schlüsselkandidate:
{TAG, STUNDE, RAUM}; {TAG, STUNDE, PROFESSOR}; {TAG, STUNDE, GRUPPE}
Ein Fremdschlüssel ist eine Menge von Attributen aus einer Relation R1, die auf einem Tupel aus einer anderen Relation R2 verweist (ein logischer Pointer oder Zeiger). Der Fremdschlüssel muss einem Primärschlüssel in der zweiten Relation entsprechen. Die zwei Relationen R1 und R2 sind nicht notwendigerweise verschieden.

R1 **Fremd- R2**
Schlüssel **schlüssel**

Beispiel:
KLASSEN [Id, Abteilung]
STUDENTEN [Id, Name, Klasse, Geburtsdatum]



Für den obigen Beispiel gibt es eine Beziehung zwischen der Relation KLASSEN (die auch Vatertabelle/übergeordnete Relation/parent relation heißt) und der Relation STUDENTEN (Kindtabelle/untergeordnete Relation) durch die Gleichung KLASSEN.Id = STUDENTEN.Klasse. Eine Klasse (aus der Relation KLASSEN) identifiziert durch den Id steht in Beziehung zu allen Studenten (gespeichert in der Relation STUDENTEN), die zu der Klasse gehören.
Mit Hilfe von Fremdschlüsseln kann man 1:N Beziehungen zwischen Entitäten modellieren: eine Klasse steht in Beziehung mit mehreren Studenten, und ein Student steht in Beziehung mit einer einzigen Klasse.
Fremdschlüsseln können auch dafür benutzt werden, eine M:N Beziehung zu modellieren.
Zum Beispiel, gibt es eine M:N Beziehung zwischen der Relationen Vorlesungen und Studenten, da es mehrere Studenten gibt die eine Vorlesung hören und ein Student mehrere Vorlesungen während eines Semesters hören kann. Eine M:N Beziehung wird in einer zusätzlichen Tabelle modelliert, wie zum Beispiel Studienverträge in diesem Fall:



Integritätsregeln sind Bedingungen, die für jede Instanz der Datenbank erfüllt werden müssen. Integritätsregeln werden beim Erstellen des Schemas festgelegt und beim

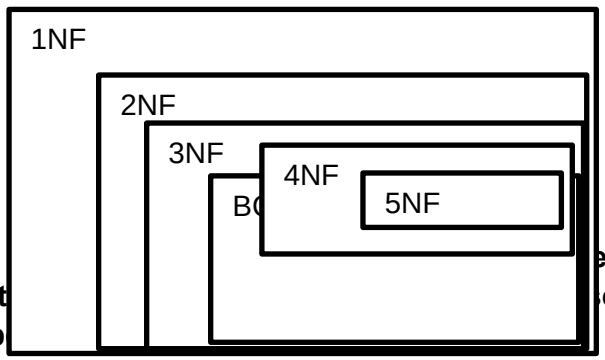
Ändern der Relationen überprüft. Eine Instanz der Datenbank, die alle Integritätsregeln erfüllt heißt legale Instanz.
Es gibt Integritätsregeln in Bezug auf Attribute, Relationen oder Beziehungen zwischen den Relationen:

- **Integritätsregeln für Attribute**
  - **Not Null – das Attribut darf nicht den Wert „unbekannt“ haben**
  - **Primary Key – das aktuelle Attribut ist Primärschlüssel für die Relation**
  - **Unique – das aktuelle Attribut ist ein Kandidatschlüssel**
  - **Check (Bedingung) – einfache logische Bedingungen, die erfüllt werden müssen damit die Datenbank konsistent bleibt**
  - **Foreign Key REFERENCES parent_table [(attribute_name)][On Update action] [On Delete action] – das aktuelle Attribut ist eine Referenz auf ein Tupel aus einer anderen Relation**
- **Integritätsregeln für Relationen**
  - **Primary Key(Attributenliste) – definiert einen zusammengesetzten Primärschlüssel für die Relation**
  - **Unique(Attributenliste) – definiert einen zusammengesetzten Kandidatschlüssel für die Relation**
  - **Check(Bedingung) – eine Bedingung, die mehr als ein Attribut enthält**
  - **Foreign Key foreign_key(attribute_list) REFERENCES parent_table [(attribute_list)][On Update action] [On Delete action] – definiert einen zusammengesetzten Fremdschlüssel**

2.1.2. Die ersten drei Normalformen einer Relation

Im Allgemeinen, gibt es mehrere Möglichkeiten die Daten durch Relationen darzustellen (relationales Modell). Redundanz ist oft die Ursache von Schemata Probleme**: redundante Speicherung, Einfüge-, Lösch- und Update-Anomalie. Integritätsregeln** helfen ein schlechtes Schema zu identifizieren **und Verfeinerungen vorzuschlagen.**
Wenn sich eine Relation in einer bestimmten Normalform befindet, dann wissen wir, dass bestimmte Probleme nicht vorkommen können. Die am meisten benutzten Normalformen heutzutage sind: 1NF, 2NF, 3NF, BCNF, 4NF und 5NF. Zwischen den Normalformen gilt folgende Relation:

5NF ⊆ 4NF ⊆ BCNF ⊆ 3NF ⊆ 2NF ⊆ 1NF



Falls sich eine Relation efindet, dann kann diese in mehreren Relat chten Normalform sind. Der Projektion Op zerlegen und der Natürlicher Verbund Operator (Join) um die Relation wiederherzustellen.

Definition. Sei $\alpha = \{A_{i_1}, A_{i_2}, ..., A_{i_p}\}$ eine Teilmenge von Attributen(Spalten) aus der Relation R[A_1, …, A_n]. Die Projektion der Attribute α einer Relation R ist definiert als die Relation:

$$R'[A_{i_1}, A_{i_2}, ..., A_{i_p}] = \prod_{\alpha}(R) = \prod_{\{A_{i_1}, A_{i_2}, ..., A_{i_p}\}}(R),$$

wobei alle Elemente aus R' verschieden (distinkt) sind und:

$$\forall r = (a_1, a_2, ..., a_n) \in R \Rightarrow \prod_{\alpha}(r) = r[\alpha] = (a_{i_1}, a_{i_2}, ..., a_{i_p}) \in R' .$$

Definition. Gegeben seien zwei Relationen $R[\alpha, \beta]$, $S[\beta, \gamma]$, auf die Attributen Mengen α, β, γ, $\alpha \cap \gamma = \emptyset$, durch den **natürlichen Join (Verbund)** der zwei Relationen erhalten wir folgende Relation:

$$R \bowtie S[\alpha, \beta, \gamma] = \{(\Pi_{\alpha}(r), \Pi_{\beta}(r), \Pi_{\gamma}(s)) \mid r \in R, s \in S \text{ und } \Pi_{\beta}(r) = \Pi_{\beta}(s)\} .$$

Eine Relation **R** kann in mehrere (neue) Relationen zerlegt werden: R_1 , R_2 ,...,R_m. Diese Zerlegung nennen wir **verlustlos (lossless-join)** falls $R = R_1 \bowtie R_2 \bowtie ... \bowtie R_m$, d.h. die in R erhaltene Information muss über den natürlichen Verbund der Relationen R_1, ..., R_m rekonstruierbar sein und keine neue Tupeln dürfen in dem Join-Ergebnis auftauchen.

Beispiel von einer Zerlegung, die nicht verlustlost ist:

Sei die Relation **Studienvertrage[Student, Professor, Kurs],**

und zwei neue Relationen erhalten als die Projektion von **Studienvertrage** auf folgende Attributenmengen {Student, Professor} und {Professor, Kurs}. Wenn die ursprüngliche Relation folgende Tupeln enthält:

R	Student	Professor	Kurs
r1	s1	p1	c1
r2	s2	p2	c2
r3	s1	p2	c3

dann erhalten wir nach der Projektion folgende zwei Relationen:

SP	Student	Professor
r1	s1	p1
r2	s2	p2
r3	s1	p2

PK	Professor	Kurs
r1	p1	c1
r2	p2	c2
r3	p2	c3

Wenn wir den natürlichen Join der neuen Relationen berechnen, dann erhalten wir folgende Relation:

SP ⋈ PK	Student	Professor	Kurs
r1	s1	p1	c1
r2	s2	p2	c2
?	s2	p2	c3
?	s1	p2	c2
r3	s1	p2	c3

Das Ergebnis des natürlichen Joins SP ⋈ PK enthält zwei neue Tupeln welche nicht Teil der ursprünglichen Relation R sind, d.h. die Zerlegung ist nicht verlustlos.

Bemerkung. Ein einfaches Attribut ist jedes Attribut einer Relation und ein zusammengesetztes Attribut ist eine Menge von Attributen (wenigstens zwei) derselben Relation.
In Anwendungen kann es nötig sein, dass manche Attribute mehrere Werte haben für dasselbe Objekt. Diese Attribute nennen wir mehrwertige (repetitive) Attribute.

Beispiel 4. Sei die Relation:

STUDENTEN [NAME, GEBURTSJAHR, GRUPPE, KURS, NOTE]
mit NAME als Primärschlüsse.
In diesem Fall ist die Menge {KURS, NOTEN} ein zusammengesetztes mehrwertiges Attribut. Die Relation kann folgende Werte enthalten:

NAME	GEBURTSJAHR	GRUPPE	KURS	NOTE
Popa Mihai	1998	221	Datenbanken	10
			Betriebssysteme	9
			Wahrscheinlichkeitstheorie	8
Muresan	1999	222	Datenbanken	8
Andrei			Betriebssysteme	7
			Wahrscheinlichkeitstheorie	10
			Projekt	9

Beispiel 5. Sei die Relation

BUCH [ISBN, AUTHORNAME, TITEL, VERLAG, JAHR, SPRACHE, SCHLÜSSELWÖRTER] mit ISBN als Primärschlüssel.
Die Attributen AUTHORNAME und SCHLÜSSELWÖRTER sind mehrwertig.

Mehrwertige Attribute können zu Probleme führen. Darum sucht man im Praxis Lösungen um mehrwertige Attribute zu vermeiden, ohne aber Daten zu verlieren.
Bemerkung. Sei R[A] eine Relation, A eine Menge von Attributen und α ein mehrwertiges Attribut (einfach oder zusammengesetzt). Dann kann R in zwei Relationen zerlegt werden, die keine mehrwertigen Attribute enthalten. Wenn C ein Schlüssel der Relation R ist, dann sind die zwei Relation aus der Zerlegung:

$$R'[C \cup \alpha] = \prod_{C \cup \alpha}(R) \text{ und } R''[A - \alpha] = \prod_{A-\alpha}(R).$$

Beispiel 6. Relation STUDENTEN aus dem Beispiel 4 kann folgendermaßen zerlegt werden:

ALLGEMEINE_DATEN [NAME, GEBURTSJAHR, GRUPPE**]**

ERGEBNISSE [NAME, KURS, NOTE**]**

Beispiel 7. Relation **BUCH aus Beispiel 5 kann in 3 Relationen zerlegt werden (BUCH hat 2 mehrwertige Attribute):**

BUCH [ISBN, TITEL, VERLAG, JAHR, SPRACHE]

AUTHOREN [ISBN, AUTHORNAME]

SCHLÜSSELWÖRTER [ISBN, SCHLÜSSELWORT]

Bemerkung. Wenn ein Buch keine Authoren oder Schlüsselwörter hat, dann gibt es trotzdem ein entsprechendes Tupel in den Relationen AUTHOREN und SCHLÜSSELWÖRTER, die aber den Wert NULL haben für Authorename und

Schlüsselwort. Falls diese Tupel gelöscht werden, dann kann man die Relation BUCH durch den natürlichen Join nicht mehr wiederherstellen (man müsste Outer Join benutzen).

Definition. Eine Relation ist in der ersten Normalform, wenn alle Attribute der Relation **atomar** sind. Zusammengesetzte und mehrwertige Attribute sind nicht erlaubt.

Die meisten Dantebankmanagement Systeme erlauben nur die Erstellung von Relationen in 1NF. Es gibt aber auch Systeme, wo mehrwertige Attributen erlaubt sind (z.B. Oracle, wo eine Spalte ein *Objekt* oder eine *Datenmenge/data collection* ist).

Die nächsten zwei Normalformen werden mit Hilfe von funktionalen Abhängigkeiten definiert. Funktionale Abhängigkeiten beschreiben Beziehungen zwischen den Attributen. Der Entwerfer der Datenbank definiert die funktionale Abhängigkeiten für alle Beziehungen in der Datenbank, indem er die Bedeutung der Daten versteht. Alle Einfüge-, Lösch- und Änderungsoperationen müssen die funktionale Abhängigkeiten bewahren.

Definition. Gegeben sei die Relation $R[A_1, A_2, ..., A_n]$ und zwei Attributenmengen $\alpha, \beta \subset \{A_1, A_2, ..., A_n\}$. Die Attributenmenge β ist von α **funktional abhängig**, mit der Notation $\alpha \to \beta$, wenn und nur wenn zu jedem möglichen Wert in α genau ein Wert in β gehört (diese Bedingung muss für jede Ausprägung der Relation gültig sein, auch wenn sich die Daten ändern). Wenn derselbe Wert für α in mehreren Tupeln vorkommt, dann müssen diese Tupeln denselben Wert auch für β haben. Das heißt:

$$\text{aus } \prod_{\alpha}(r) = \prod_{\alpha}(r') \text{ folgt } \prod_{\beta}(r) = \prod_{\beta}(r').$$

Bemerkung. Eine funktionale Abhängigkeit ist eine Regel, die für jede Ausprägung der Relation gelten muss. Daten werden eingefügt oder geändert nur dann, wenn die funktionale Abhängigkeit immer noch gültig bleibt.

Die Existenz der funktionalen Abhängigkeiten in einer Relation bedeutet, dass die Relation Redundanzen enthält wenn „Paare" mit denselben Werten mehrmals gespeichert werden. Schauen wir uns das nächste Beispiel an:

Beispiel 8. KLAUSUR [<u>StudentenName, KursName</u>, Note, Professor]

{StudentenName, KursName} ist Primärschlüssel. Da eine Vorlesung mit einem einzigen Professor in Beziehung steht, haben wir folgende fkt. Abh.: {KursName}→ {Professor}. Umgekehrt gilt die fkt. Abh. nicht, da ein Professor mehrere Vorlesungen hält.

Klausur	StudentenName	KursName	Note	Professor
1	Alb Ana	Mathematik	10	Rus Teodor
2	Costin Constantin	Geschichte	9	Popa Horea
3	Alb Ana	Geschichte	8	Popa Horea
4	Enisei Elena	Mathematik	9	Rus Teodor
5	Frisan Florin	Mathematik	10	Rus Teodor

Wenn die funktionale Abhängigkeit in dieser Relation bewahrt wird können folgende Probleme vorkommen:

- **zusätzliches Speicherplatz wird gebraucht: z.B. speichern wir dreimal die Information, dass Prof. Rus Teodor die Vorlesung Mathematik liest.**
- **Update Anomalie:** Eine Aktualisierung kann zu Inkonsistenzen führen, wenn die Änderung nicht in allen betroffenen Datensätze durchgeführt wird. In diesem Beispiel, wenn wir den Namen des Mathematik-Professors ändern müssen, dann

müssen wir 3 Tupeln aktualisieren, sonst wird die funktionale Abhängigkeit nicht mehr bewahrt.

- **Einfüge Anomalie:** Wir können kein neues Tupel einfügen, wenn wir die Werte für die andere entspreche Attribute nicht kennen (Attribute, die in einer funktionalen Abhängigkeit beteiligt sind, dürfen keine NULL-Werte haben). D.h. wir können eine neue Vorlesung nicht einfügen bevor wir den Professor kennen, der diese Vorlesung liest.
- **Lösch Anomalie**: Beim Löschen der Informationen zu einer Entität können Informationen zu einer anderen Entität oder zu einer Beziehung ungewollt verloren gehen. Z.B. wenn wir Tupel 2 und 3 löschen, dann verlieren wir die Beziehung zwischen Professor *Popa Horea* und der Vorlesung *Geschichte*.

Diese Anomalien werden von funktionalen Abhängigkeiten zwischen Mengen von Attributen verursacht (und von schlechtem Schemadesign). Anomalien können vermieden werden, indem die Relation zerlegt wird, sodass die Attribute, die in der funktionalen Anhängigkeit beteiligt sind, sich in unterschiedlichen Relationen befinden. Um das zu erreichen muss die ursprüngliche Relation verlustlos zerlegt werden. Eine solche verlustlose Zerlegt sollte in der Entwurf Phase stattfinden, wann auch die funktionalen Abhängigkeiten identifiziert werden.

Bemerkung. Man kann folgende Eigenschaften für funktionale Abhängigkeiten beweisen:

1. **Wenn C ein Schlüssel für** $R[A_1, A_2, ..., A_n]$ ist, dann gilt $C \to \beta, \forall \beta \subset \{A_1, A_2, ..., A_n\}$.
2. Wenn $\beta \subseteq \alpha$, dann heißt $\alpha \to \beta$ eine **triviale funktionale Abhängigkeit** oder **Reflexivität.**

$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\beta \subset \alpha}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow \alpha \to \beta$$

3. Wenn $\alpha \to \beta$, dann gilt $\gamma \to \beta, \forall \gamma$ cu $\alpha \subset \gamma$.

$$\Pi_\gamma(r_1) = \Pi_\gamma(r_2) \underset{\alpha \subset \gamma}{\Rightarrow} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\alpha \to \beta}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow \gamma \to \beta$$

4. Wenn $\alpha \to \beta$ und $\beta \to \gamma$, dann gilt $\alpha \to \gamma$, d.h. **funktionale Abhängigkeiten** sind transitiv.

$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\alpha \to \beta}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \underset{\beta \to \gamma}{\Rightarrow} \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \Rightarrow \alpha \to \gamma$$

5. Wenn $\alpha \to \beta$ und $\gamma \subset A$, dann gilt $\alpha\gamma \to \beta\gamma$, wobei $\alpha\gamma = \alpha \cup \gamma$.

$$\Pi_{\alpha\gamma}(r_1) = \Pi_{\alpha\gamma}(r_2) \Rightarrow \left| \begin{array}{c} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \Rightarrow \Pi_\beta(r_1) = \Pi_\beta(r_2) \\ \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \end{array} \right. \Rightarrow \Pi_{\beta\gamma}(r_1) = \Pi_{\beta\gamma}(r_2)$$

Definition. Ein Attribut A (einfach oder zusammengesetzt, also ein Attribut oder eine Menge von Attributen) heißt prim, wenn es ein Schlüssel (Kandidatenschlüssel) C gibt, sodass A in C enthalten ist. Ein Attribut heißt **nicht prim** oder **Nichtschlüssel-Attribut,** wenn es in keinem Schlüssel der Relation enthalten ist.

Definition. Sei die Relation $R[A_1, A_2, ..., A_n]$ und die Attributenmengen $\alpha, \beta \subset \{A_1, A_2, ..., A_n\}$. Das Attribut β ist **voll funktional abhängig** von α wenn β funktional abhängig von α ist

(also $\alpha \to \beta$) und zusätzlich gibt es keine echten Teilmengen der Attributenmenge α von der β funktional abhängig ist (d.h. $\forall \gamma \subset \alpha, \gamma \to \beta$ gilt nicht).

Bemerkung. Wenn β nicht voll funktional abhängig von α ist, dann ist α ein zusammengesetztes Attribut (also eine Menge von Attributen).

Definition. Eine Relation ist in **zweiter Normalform (2 NF)** genau dann wenn:

- **die Relation in 1NF ist und**
- jedes **Nichtschlüssel-Attribut** A ∈ R **voll funktional abhängig ist von jedem Kandidatenschlüssel der Relation**

Bemerkung. Wenn eine Relation in 1NF ist, aber nicht in 2NF, dann hat die Relation ein zusammengesetzter Schlüssel (eine Relation die nicht in 2NF ist hat eine fkt. Abh. $\alpha \to \beta$ sodass α Teil eines Schlüssels ist).

Das allgemeine Zerlegung Verfahren kann folgendermaßen definiert werden. Sei die Relation $R[A_1, A_2, ..., A_n]$ und ein Schlüssel $C \subset A = \{A_1, A_2, ..., A_n\}$. Man nimmt an, dass es ein Nichtschlüssel Attribut $\beta \subset A$, $\beta \cap C = \emptyset$, gibt, sodass die Abhängigkeit $C \to \beta$ nicht voll ist, d.h. $\exists \alpha \subset C$ mit $\alpha \to \beta$. Die funktionale Abhängigkeit $\alpha \to \beta$ gibt uns dann die Zerlegung:

$$R'[\alpha \cup \beta] = \prod_{[\alpha \cup \beta]}(R) \text{ und } R''[A - \beta] = \prod_{A-\beta}(R).$$

In Beispiel 8 hatten wir die Relation:

KLAUSUR[<u>StudentenName, KursName</u>, Note, Professor],

mit dem Primärschlüssel {StudentenName, KursName} und mit der funktionalen Abhängigkeit KursName}→ {Professor}. Man merkt, dass das Attribut *Professor* von dem Schlüssel nicht voll funktional abhängig ist, also ist die Relation KLAUSUR nicht in 2NF. Man kann aber die Relation KLAUSUR in folgende zwei Relationen zerlegen:

KATALOG [StudentenName, KursName, Note]

KURSE [KursName, Professor].

Beispiel 9. Für das Verwalten der Studienverträge wird folgende Relation benutzt:

VERTRÄGE [Nachname, Vorname, <u>CNP, KursId</u>, KursName]

Der Schlüssel der Relation ist {CNP, KursId}. In dieser Relation kann man folgende funktionale Abhängigkeiten identifizieren:

{CNP}→ {Nachname, Vorname}* und *{KursId}→{KursName}

Um diese funktionale Abhängigkeiten zu behandeln, kann man die ursprüngliche Relation in folgende Subrelationen zerlegen:

STUDENTEN [CNP, Nachname, Vorname]

KURSE [KursId, KursName]

VERTRÄGE [CNP, KursId]

Um die dritte Normalform zu definieren müssen wir transitive Abhängigkeiten einführen.

Definition. Ein Attribut Z ist transitiv abhängig von dem Attribut X wenn ∃ **Y** sodass X $\to$ Y, Y $\to$ Z, wobei Y $\to$ X nicht gilt und Z in $X \cup Y$ nicht enthalten ist.

Definition. Eine Relation ist in der **dritten Normalform (3NF)**, wenn sie in der **2NF** ist und **kein Nichtschlüsselattribut von einem Schlüsselkandidaten transitiv abhängt.**

Wenn C ein Schlüssel ist und β ein Attribut, das von C transitiv abhängig ist, dann gibt es ein Attribut α für welches: $C \to \alpha$ (gilt immer, da C Schlüssel ist) und $\alpha \to \beta$. Da die Relation in 2NF ist, folgt dass β voll funktional von C ist, also $\alpha \not\subset C$. Wir können schlussfolgern, dass eine Relation die in 2NF, aber nicht in 3NF ist, eine funktionale Abhängigkeit der Form $\alpha \to \beta$ enthält, wobei α ein Nichtschlüsselattribut (nicht prim) ist. Die Abhängigkeit kann behandelt werden indem die Relation zerlegt wird.

Beispiel 10. Die Noten der Studenten bei der Linzenzprüfung werden in folgende Relation gespeichert:

LIZENZPRÜFUNG [StudentenName, Note, Betreuer, Abteilung].

Da es in der Relation höchstens ein Tupel für einen Studenten geben kann, können wir *StudentenName* als Primärschlüssel auswählen. Basierend auf der Bedeutung der Attribute kann man folgende funktionale Abhängigkeit identifizieren:

{*Betreuer*} → {*Abteilung*}.

Mit dieser funktionalen Abhängigkeit ist die Relation nicht in 3NF. Diese kann aber folgendermaßen zerlegt werden:

ERGEBNISSE [StudentenName, Note, Betreuer]
BETREUERN [Betreuer, Ableitung]

Beispiel 11. Die Adressen der Personen werden in folgende Relation gespeichert:

ADRESSEN [<u>CNP</u>, Nachname, Vorname, Postleitzahl, Stadt, Straße, Nr]

Der Schlüssel der Relation ist {*CNP*}. Wir identifizieren folgende funktionale Abhängigkeit:
{*Postleitzahl*} → {*Stadt*}.
Wegen dieser fkt. Abh. ist die Relation ADRESSEN nicht in 3NF, also man muss diese zerlegen.

Beispiel 12. Gegeben sei folgende Relation, welche der Stundenplan der Prüfungen speichert:

PRÜFUNG_PLAN [Datum, Zeit, Professor, Raum, Gruppe],

mit folgenden Einschränkungen:

1. **ein Student muss höchstens eine Prüfung pro Tag haben, also ist {*Gruppe, Datum*} Kandidatschlüssel**
2. **ein Professor hat eine einzige Prüfung mit einer einzigen Gruppe an einem Datum und Zeit, also ist {*Professor, Datum, Stunde*} Kandidatschlüssel**
3. **an einem bestimmten Zeitpunkt und in einem Raum findet eine einzige Prüfung statt, also ist {*Raum, Datum, Zeit*} Kandidatschlüssel**
4. **ein Professor ändert den Raum an einem Tag nicht (er kann aber mehrere Prüfungen in demselben Raum haben), also gilt {*Professor, Datum*} → {*Raum*}**

Alle Attribute der Relation PRÜFUNG_PLAN sind in einem Schlüssel enthalten, also gibt es keine Nichtschlüsselattribute. Man merkt, dass diese Relation in 3NF ist. Um aber, ähnlich wie vorher, die funktionalen Abhängigkeiten zu behandeln, wird eine neue Normalform eingeführt:

Definition. Eine Relation ist in Boyce-Codd Normalform (3NF), wenn alle für alle Abhängigkeiten $\alpha \to \beta$ gilt:

- $\beta \subseteq \alpha$ (die fkt. Abh. ist trivial) **oder**
- α enthält einen Schlüssel von R (α ist ein Superschlüssel)

In Beispiel 12 kann die Relation in folgende Relationen PRÜFUNG_PLAN zerlegt werden:

PRÜFUNG_PLAN [Datum, Professor, Zeit, Gruppe]

RAUM_PLAN [Professor, Datum, Raum]

Nach der Zerlegung erhalten wir zwei Relationen, die in BCNF sind. Leider, wurde auch die Einschränkung vom Punkt 3 entfernt, also ist {*Raum, Datum, Zeit*} kein Schlüssel mehr (die Attribute befinden sich nicht mehr in derselben Relation). Wenn diese Einschränkung trotzdem nötig wäre, dann könnte diese mit einer anderen Methode überprüft werden (in der Anwendung).

2.2. Datenbanken Abfragen mit Hilfe der relationalen Algebra

Im Folgenden findet ihr eine Liste von Bedingungen, die in der Definition der relationalen Operatoren vorkommen:

1. **Um zu überprüfen, dass ein Attribut eine einfache Bedingung erfüllt, kann dieses mit einem Wert verglichen werden:**

   AttributName relationalen_Operator Wert

2. **Eine Relation mit einer einzigen Spalte kann als eine Menge von Elementen betrachtet werden. Die nächste Bedingung prüft ob ein bestimmtes Wert in einer Menge enthalten ist:**

   AttributName $\left\{ \begin{matrix} IS\ IN \\ IS\ NOT\ IN \end{matrix} \right\}$ **Relation_mit_einer_Spalte**

3. **Zwei Relationen (die als Menge von Tupeln oder Zeilen betrachten werden) können mit folgenden Operatoren verglichen werden: Gleichheit, verschieden, enthalten oder nicht enthalten. Folgende Operatoren können auf Relationen mit derselben Anzahl von Spalten benutzt werden:**

   Relation $\left\{ \begin{matrix} IS\ IN \\ IS\ NOT\ IN \\ = \\ <> \end{matrix} \right\}$ **Relation**

4. **Bedingungen können mit boolesche Operatoren zusammengesetzt werden:**

   (Bedingung)

   NOT Bedingung

   Bedingung1 AND Bedingung2

   Bedingung1 OR Bedingung2

   wobei *Bedingung*, *Bedingung1, Bedingung2* jede Bedingung der Form 1-4 sein können.

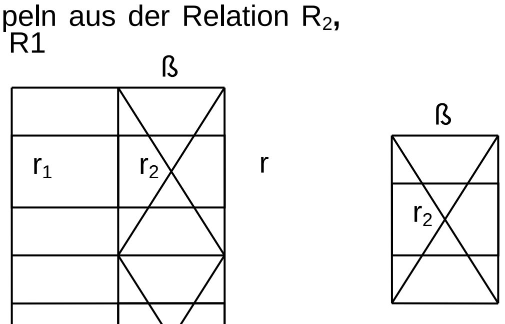
In Bedingungen der 1. Form haben wir den Begriff *Wert* benutzt. Dieser hat einen der folgenden Typen:

- **Attribut_Name – gibt den Wert eines Attributes aus dem aktuellen Tupel an. Wenn der Attributverweis vieldeutig ist (das Attribut ist in mehreren Relationen enthalten), dann nimmt man die Relationenname auch dazu: Relation.Attribut**
- **Ausdruck (expression) – ein Ausdruck enthält Werte und Operatoren, die basierend auf die aktuellen Zeilen der befragten Relation ausgewertet werden**

- **COUNT(*) FROM Relation – gibt die Anzahl der Tupeln einer Relation an**
- $\left\{\begin{matrix} COUNT \\ SUM \\ AVG \\ MAX \\ MIN \end{matrix}\right\}$ **([DISTINCT] Attribut_Name) – identifiziert einen Wert aus einer Menge von Werten in Bezug auf allen Tupeln aus der Relation. Der Wert wird mithilfe aller Attributenwerte (aus allen Tupeln), oder nur mit der Menge der Attributenwerte ohne Duplikate (wenn man DISTINCT benutzt), berechnet. Das Ergebnis enthält die Anzahl der Werte (für COUNT), die Summe der Werte (für SUM; alle Werte müssen numerisch sein), den Mittelwert (für AVG; alle Werte müssen numerisch sein), das Maximum (für MAX) oder das Minimum (für MIN).**

Für das Abfragen relationalen Datenbanken kann man folgende Operatoren benutzen:

- **Selektion (oder horizontale Projektion) auf eine Relation R – gibt eine Relation mit demselben Schema wie R zurück. Aus R wählt man nur die Tupel (Zeilen), welche eine bestimmte Bedingung C erfüllen. Für Selektion verwendet man die Notation: σ_C(R).**
- **Projektion (oder vertikale Projektion) – gibt eine neue Relation zurück, die als Attributenmenge eine Teilmenge α der Attribute aus R hat. Die Menge α kann zu einer Menge von Ausdrücken erweitert werden, wo auch die Namen der neuen Spalten bestimmt werden müssen. Für Projektion verwendet man die Notation: π_α(R)**
- **Kartesisches Produkt zweier Relationen, mit der Notation $R_1 \times R_2$, gibt eine neue Relation zurück, die alle Attribute der Relationen** R_1 und R_2 enthält (ein Tupel aus der neuen Relation wir als Verkettung zweier Tupel aus der ersten und zweiten Relation erhalten)
- **Vereinigung, Differenz und Durchschnitt zweier Relationen:** $R_1 \cup R_2$, $R_1 - R_2$, $R_1 \cap R_2$. Die zwei Relationen müssen dasselbe Schema haben.
- **Join-Operatoren:**
    - **θ-Join (Theta-Verbund, conditional join)**, mit der Notation $R_1 \bowtie_c R_2$ – gibt die Tupeln des kartesischen Produkts zurück, die eine bestimmte Bedingung erfüllen. D.h. $R_1 \bowtie_c R_2 = \sigma_c (R_1 \times R_2)$
    - **Natürlicher Verbund**, mit der Notation $R_1 \bowtie R_2$ – gibt eine neue Relation zurück, welche als Attribute die Vereinigung der Attribute der zwei Relationen enthält und als Tupeln die Paare der Tupel der zwei Relationen, die den gleichen Wert für die gemeinsamen Attribute haben. Wenn die zwei Relationen das Schema $R_1[\alpha], R_2[\beta]$ haben und $\alpha \cap \beta = \{A_1, A_2, ..., A_m\}$, dann berechnet man den natürlichen Verbund folgendermaßen:
    $$R_1 \bowtie R_2 = \pi_{\alpha\cup\beta}\left(R_1 \bowtie_{R_1.A_1 = R_2.A_1 \ and ... \ and \ R_1.A_m = R_2.A_m} R_2\right)$$
    - **Left Outer Join**, mit der Notation R_1 **⟕$_c$** R_2, gibt eine neue Relation zurück, welche als Attribute die Vereinigung der Attribute der zwei Relationen enthält. Als Tupeln enthält der left outer Join alle Tupeln aus dem θ-Join $R_1 \bowtie_c R_2$ und zusätzlich die Tupel aus Relation R_1, **die nicht im** θ-Join **enthalten sind, zusammen mit Null-Werte für die Attribute entsprechend der Relation** R_2.
    - **Right Outer Join**, mit der Notation R_1 **⟖$_c$** R_2**, ähnelt dem left outer Join Operator, enthält aber zusätzlich zum** θ-Join Tupeln aus der Relation R_2**, die kein Paar in** R_1 **haben.**



- **Division hat die Notation $R_1 \div R_2$** (oder R_1 / R_2).
  Nehmen wir an, dass die Relation R_1 zwei Mengen von Attributen *x* und *y* enthält und R_2 enthält nur die Attributenmenge *y* (d.h. *y* ist die gemeinsame Menge von Attributen der zwei Relationen).
  $R_1 \div R_2$ enthält alle *x* Tupeln so dass für jedes *y* Tupel in R_2, ein *xy* Tupel in R_1 existiert.

  Anders gesagt, wenn eine Menge *y* von Attributen, die mit einem *x* Wert in R_1 **verbunden ist, alle *y* Werte aus** R_2 enthält, dann sind die *x* Werte in $R_1 \div R_2$ enthalten (siehe Figur).

Eine wichtige Aufgabe im Bereich der relationalen Algebra ist eine Untermenge von unabhängigen relationalen Operatoren zu bestimmen. Ein Menge M von Operatoren ist unabhängig wenn ein Operator nicht mithilfe der anderen Operatoren dargestellt werden kann.
Für relationalen Algebra ist {$\sigma, \pi, \times, \cup, -$}eine unabhängige Menge von Operatoren. Alle anderen Operatoren können unter Verwendung eines der folgenden Regeln erhalten werden:

- $R_1 \cap R_2 = R_1 - (R_1 - R_2)$;
- $R_1 \bowtie_c R_2 = \sigma_c(R_1 \times R_2)$;
- $R_1[\alpha]$, $R_2[\beta]$, und $\alpha \cap \beta = \{A_1, A_2, ..., A_n\}$, dann
  $R_1 \bowtie R_2 = \pi_{\alpha \cup \beta}(R_1 \bowtie_{R_1.A_1 = R_2.A_1 \, and \ldots and \, R_1.A_m = R_2.A_m} R_2)$;
- Sei $R_1[\alpha]$, $R_2[\beta]$, und $R_3[\beta]$ = (null, ..., null), $R_4[\alpha]$ = (null, ..., null).

  $$R_1 \overset{\leftarrow}{\bowtie}_c R_2 = (R_1 \bowtie_c R_2) \cup [R_1 - \pi_\alpha(R_1 \bowtie_C R_2)] \times R_3 .$$

  $$R_1 \overset{\rightarrow}{\bowtie}_c R_2 = (R_1 \bowtie_c R_2) \cup R_4 \times [R_2 - \pi_\beta(R_1 \bowtie_C R_2)].$$

- Sei $R_1[\alpha]$, $R_2[\beta]$, mit $\beta \subset \alpha$. Dann gilt $r \in R_1 \div R_2$ wenn $\forall r_2 \in R_1 \div R_2$, $\exists r_1 \in R_1$ sodass: $\pi_{\alpha-\beta}(r_1) = r$ und $\pi_\beta(r_1) = r_{2n}$

  Es folgt, dass **r** in $\pi_{\alpha-\beta}(R_1)$ enthalten ist. In $\left(\pi_{\alpha-\beta}(R_1)\right) \times R_2$ sind alle Elemente, die ein Teil in $\pi_{\alpha-\beta}(R_1)$ und der andere Teil in R_2 haben. Aus dem Ergebnis eliminieren wir R_1 und was bleibt sind die Elemente, die ein Teil in $\pi_{\alpha-\beta}(R_1)$ und kein Teil in $\pi_\beta(R_1)$ haben. Es folgt:

  $$R_1 \div R_2 = \pi_{\alpha-\beta}(R_1) - \pi_{\alpha-\beta}\left(\left(\pi_{\alpha-\beta}(R_1)\right) \times R_2 - R_1\right).$$

Zusätzlich zu der Operatoren Liste erwähnen wir ein paar nützliche Operatoren:

- **Zuweisungsoperation – das Ergebnis einer Abfrage kann temporär in einer Relation R gespeichert werden. Die Attribute in der neuen Relation R können auch angegeben werden.**
  R[Liste]: =expression **oder** **R** ←expression
- **Duplikateliminierung aus einer Relation:** $\delta(R)$
- Sortieren der Tupeln einer Relation: $s_{\{Liste\}}(R)$

- Erweiterte Projektion, wo arithmetische Funktionen als Projektionbedingung benutzt werden können: $\pi_{F1,\ldots,Fn}(R)$
- Gruppierung:

$$_{G_1,G_2,\ldots,G_n}\vartheta_{F_1(A_1),F_2(A_2),\ldots,F_n(A_n)}(R), \text{ wobei}$$

$G_1,G_2,\ldots,G_n$ – eine Liste von Attributen worauf wir gruppieren wollen
F_i – Aggregatfunktion, A_i – Name eines Attributes

2.3. Datenbanken Abfragen in SQL (Select)

SQL (Structured Query Language) wurde als relationale Datenbankabfragesprache entwickelt. SQL ist nützlich für das Verwalten der Datenbank Objekten (Tabellen, Indexe, Benutzer, Sichten, gespeicherte Prozeduren, Triggern, usw.)
Kurze Geschichte:

- 1970 – E.F. Codd formalisiert das relationale Model
- 1974 – mit Beteiligung von E.F. Codd (IBM) wurde von D.D. Chamberlin und R.F. Boyce die Abfragesprache SEQUEL (Structured English Query Language) entworfen
- 1975 – die Abfragesprache SQUARE (Specifying Queries as Relational Expressions) wurde entworfen
- 1976 – IBM entwirft eine neue SEQUEL Version, SEQUEL/2. Nach der ersten Bearbeitung (revision) entsteht SQL.
- 1986 – SQL wird zum ANSI-Standard (American National Standards Institute)
- 1987 – SQL wird von ISO (International Standards Organization) angenommen
- 1989 – die Erweiterung SQL89 (oder SQL1) wird herausgegeben
- 1992 – nach Revision wird diese zur SQL92 (oder SQL2)
- 1999 – SQL wird mit Objekt-orientierten Konzepte aktualisiert und es wird zur SQL 3 (oder SQL1999)
- 2003- neue Datentypen und Funktionen werden eingeführt in SQL2003

Die **SELECT Klausel** wird für Datenbank Abfragen benutzt um Informationen aus der gespeicherten Daten zu erhalten. SELECT erlaubt Daten aus mehreren Datenquellen zu kombinieren. Eine SELECT Klausel kann Selektion, Projektion, kartesisches Produkt, Join, Vereinigung, Durchschnitt und Differenz Operatoren enthalten.
Die Syntax einer SELECT Anweisung ist:

$$\textbf{SELECT}\ \left[\left\{\begin{matrix}\textbf{ALL}\\ \textbf{DISTINCT}\\ \textbf{TOP N [PERCENT]}\end{matrix}\right\}\right]\left\{\overset{*}{\textbf{exp [AS field]}},\ \textbf{exp [AS field]},\ \ldots\right\}$$

FROM source1 [alias1] [, source2 [alias2] …]
[WHERE condition]
[GROUP BY field_list [HAVING condition]]

$$\left[\left\{\begin{matrix}\textbf{UNION [ALL]}\\ \textbf{INTERSECT}\\ \textbf{EXCEPT}\end{matrix}\right\}\ \textbf{SELECT\_statement}\right]$$

$$\left[\textbf{ORDER BY}\ \left\{\begin{matrix}\textbf{field}\\ \textbf{nrfield}\end{matrix}\right\}\left[\left\{\begin{matrix}\textbf{ASC}\\ \textbf{DESC}\end{matrix}\right\}\right]\left[,\ \textbf{ORDER BY}\ \left\{\begin{matrix}\textbf{field}\\ \textbf{nrfield}\end{matrix}\right\}\left[\left\{\begin{matrix}\textbf{ASC}\\ \textbf{DESC}\end{matrix}\right\}\right]\ldots\right]\right]$$

Die Anweisung wählt Daten von den **sources** in dem **FROM** Klausel aus. Um die Spalten anzugeben können wir den Tabellennamen oder einen **Alias** benutzen, der auch in der **FROM** Klausel angegeben wird. Wenn wir einen Alias für die Tabellen definieren, dann können wir in dieser Anweisung die Tabellennamen nicht mehr benutzen, nur den Alias.
Source kann folgende Werte nehmen:

1. Eine **Tabelle** oder ein **Sicht** aus der Datenbank
2. (**select_statement**) - eine andere Select-Anweisung
3. **Join Anweisung**:
    - **source1 [alias1] join_operator source2 [alias2] ON link_condition**
    - **(join_expression)**

Eine **einfache Bedingung** zwischen zwei Datenquellen hat die Syntax:

[alias_source1.]field1 *operator* [alias_source2.]field2

wobei ***operator*** folgende mögliche Werte hat: =, <>, !=, <, >, <=, >=. Die zwei Werte, die verglichen werden müssen, sollen aus unterschiedlichen Tabellen kommen.
Eine **zusammengesetzte Bedingung** zwischen zwei Datenquellen hat die Syntax:

- **simple_condition [AND simple_condition] … [OR simple_condition] …**
- **(condition)**

Eine **Join Anweisung** gibt eine Tabelle aus und hat folgende Syntax:

$$\textbf{Source1}\left[\left\{\begin{matrix}\textbf{INNER}\\ \textbf{LEFT [OUTER]}\\ \textbf{RIGHT [OUTER]}\\ \textbf{FULL [OUTER]}\end{matrix}\right\}\right]\textbf{JOIN Source2 ON condition}$$

θ-Join (conditional join) aus der relationalen Algebra, mit der Notation **source1** $\bowtie_c$ **source2**, wird in SQL als **Source1 INNER JOIN Source2 on condition** geschrieben. Dieser gibt die Tupeln aus dem kartesischen Produkt zurück, welche die Bedingung aus dem ON Klausen erfüllen.
Left outer Join, geschrieben als **Source1 LEFT [OUTER] JOIN Source2 on condition,** gibt eine neue Relation zurück welche als Attribute die Vereinigung der Attribute der zwei Relationen enthält. Als Tupeln enthält der left outer Join alle Tupeln aus dem θ-Join **Source1** $\bowtie_c$ **Source2** und zusätzlich die Tupel aus Relation **Source1**, **die nicht im** θ-Join **enthalten sind, zusammen mit Null-Werte für die Attribute entsprechend der Relation Source2**.
Right outer Join, geschrieben als **Source1 RIGHT [OUTER] JOIN Source2 on condition,** gibt eine neue Relation zurück welche als Attribute die Vereinigung der Attribute der zwei Relationen enthält. Als Tupeln enthält der left outer Join alle Tupeln aus dem θ-Join **Source1** $\bowtie_c$ **Source2** und zusätzlich die Tupel aus Relation **Source2**, **die nicht im** θ-Join **enthalten sind, zusammen mit Null-Werte für die Attribute entsprechend der Relation Source1**.
Full outer Join, geschrieben als **Source1 FULL [OUTER] JOIN Source2 on condition,** wird als Vereinigung der Ergebnisse des left outer Joins und des right outer Joins berechnet.
Andere Join Anweisungen:

- Source1 **JOIN** Source2 **USING** (column_list)
- Source1 **NATURAL JOIN** Source2
- Source1 **CROSS JOIN** Source2 – gibt den kartesischen Produkt zwischen den Tupeln aus Source1 und den Tupeln aus Source2 zurück

In der **FROM** Klausel kann man Tabellen oder andere Anweisungen-Ergebnisse (die auch Tabellen sind) auswählen. Man kann mehrere Datenquellen angeben, durch Komma

getrennt. Das Ergebnis einer SELECT Anweisung, die mehrere Datenquellen in der FROM Klausel enthält, aber keine WHERE oder JOIN Klausel hat, ist das kartesische Produkt der Datenquellen (ähnlich wie CROSS JOIN).
Typischerweise gibt es eine Bedingung in der WHERE Klausel, welche die Datenquellen verbindet:

FROM source1 [, source2, …] WHERE link_condition

Zusätzlich, kann es eine weitere Bedingung geben um die Tupeln zu filtern:

WHERE link_condition AND filtering_condition

Die Bedingung **filtering_condition** kann Folgendes sein:

- Eine einfache Bedingung
- (condition)
- **not** condition
- condition1 **and** condition2
- condition1 **or** condition2

Eine einfache Bedingung hat eines der folgenden Formate:

- **expression relational_operator expression**
- **expression [NOT] BETWEEN minval AND maxval**
  um zu überprüfen ob sich ein Wert innerhalb eines Intervalls befindet oder nicht (mit NOT)
- **field (NOT) LIKE pattern**
  Pattern ist ein String, der eine Menge von möglichen Werte darstellt. In jedem Dantebankmanagement System gibt es bestimmte Schriftzeichen, welche in dem Pattern benutzt werden können, um *„jeden Schriftzeichen"(any char)* darzustellen
- **expression [NOT] IN** $\left\{ \begin{matrix} \mathbf{value}\ [\mathbf{value}]\ \ldots \\ (\mathbf{sub-selection}) \end{matrix} \right\}$
  Man prüft ob ein Wert in einer Liste von Werten oder in einer **sub-selection** enthalten ist oder nicht (NOT). **Sub-selection** ist ein Resultset von einer anderen SELECT Anweisung, die nur ein Attribut ausgibt mit demselben Typ wie der Wert (damit diese verglichen werden können).
- **field relational_operator** $\left\{ \begin{matrix} \mathbf{ALL} \\ \mathbf{ANY} \\ \mathbf{SOME} \end{matrix} \right\}$ **(sub-selection)**
  Der Wert des Attributes auf der linken Seite muss denselben Datentyp wie der Wert aus der rechten **sub-selection** haben. Die Bedingung ist erfüllt, wenn der linke Wert die Bedingung erfüllt für:
  - alle rechten Werte – **ALL**
  - wenigstens ein Wert aus der rechten Seite **– ANY** oder **SOME**
- **[NOT] EXISTS (sub-selection)**
  Die Bedingung ist ***wahr*** (oder ***falsch*** wenn **NOT** benutzt wird) wenn es wenigstens ein Tupel in der **sub-selection** gibt und ***falsch*** (oder ***wahr*** wenn **NOT** benutzt wird) wenn **sub-selection** leer ist.

Äquivalente Bedingungen:

- **„expression IN (sub-selection)"** äquivalent mit **„expression = ANY (sub-selection)"**
- **„expression NOT IN (sub-selection)"** äquivalent mit **„expression <> ALL (sub-selection)"**

Eine einfache Bedingung kann eine der folgenden Werte haben: **wahr (true), falsch (false), null.** Die Bedingung wird mit *null* ausgewertet, wenn wenigstens ein Operator aus dem Ausdruck *null* ist. Die Auswertung der Operatoren **not, and, or** wird im Folgenden beschrieben:

	True	False	Null
Not	False	True	Null

And	True	False	Null
True	True	False	Null
False	False	False	False

Or	True	False	Null
True	True	True	True
False	True	True	Null
Null	True	Null	

Null	Null	False	Null

Eine SELECT Anweisung mit „*“ in der FROM Klausel gibt eine Tabelle mit allen Attributen aus allen Datenquellen zurück. Attributen, die in mehrere Datenquellen denselben Namen haben, müssen mit den Tabellennamen oder mit dem Alias angegeben werden. Die Namen der Attribute in dem Ergebnis der Anfrage werden automatisch von dem System ausgewählt (dieselben Namen aus der Datenquellen werden behalten) oder sie können mit dem AS Klausel geändert werden.
Die Tabelle, die von einer Anfrage zurückgegeben wird enthält alle Tupeln, die die Bedingungen erfüllen, oder nur ein Teil davon, und zwar:

- ALL – alle Tupeln (default werden alle angegeben)
- DISTINCT – nur verschiedene Tupeln (keine Duplikate)
- TOP n – nur die ersten n Tupeln
- TOP n PERCENT – die ersten n% Tupeln

Die Tupel aus dem Ergebnis können aufsteigend (mit ASC) oder absteigend (mit DESC) sortiert werden mit dem **ORDER BY** Klausel. Die Spalten nach denen man sortieren will werden mit dem Namen oder mithilfe der Position (Spaltennummer) in der Liste der Attribute aus der SELECT Klausel angegeben. Die Reihenfolge der Spalten in der ORDER BY Klausel gibt die Priorität bei der Sortierung an.
Eine Menge von Tupeln aus dem Ergebnis können als ein einziges Tupel **gruppiert** werden (anstatt die ganze Gruppe wird nur ein Tupel dargestellt). Eine solche Gruppe wird durch die gemeinsame Werte der Attribute in der **GROUP BY** Klausel bestimmt. Die neue Tabelle wird automatisch nach den Werten in der GROUP BY Klausel aufsteigend sortiert und benachbarte Tupeln mit demselben Wert für diese Attribute werden durch ein einziges Tupel ersetzt. Der Tupel, der eine Gruppe ersetzt ist Teil des Endergebnisses nur dann, wenn er die Bedingung aus der **HAVING** Klausel (optional) erfüllt.
Wir können folgenden Funktionen auf die Tupel einer Gruppe benutzen:

- **AVG** $\left(\left[\left\{\begin{matrix}\mathbf{ALL}\\ \mathbf{DISTINCT}\end{matrix}\right\}\right]\ \mathbf{field}\right)$ oder **AVG([ALL] expression)**

Für eine Gruppe von Tupeln werden alle Werte (für **ALL**) oder nur die verschiedenen (ohne Duplikate für **DISTINCT**) für die Tupel oder numerische Ausdrücke in der FROM Klausel betrachtet und der **Mittelwert** wird zurückgegeben.

- **COUNT** $\left(\left\{\left[\left\{\begin{matrix}\mathbf{ALL}\\ \mathbf{DISTINCT}\end{matrix}\right\}\right]\ \mathbf{field}\right\}\right)$ $\left\{\begin{matrix}*\end{matrix}\right\}$

  Diese Funktion gibt die **Anzahl** der Tupel in einer Gruppe zurück, wobei alle Tupeln für „*“ und nur die verschiedenen für DISTINCT gezählt werden.

- **SUM** $\left(\left[\left\{\begin{matrix}\mathbf{ALL}\\ \mathbf{DISTINCT}\end{matrix}\right\}\right]\ \mathbf{field}\right)$ oder **SUM([ALL] expression)**

  Die Summe aller oder nur der verschiedenen Werte in einer Gruppe wird berechnet und zurückgegeben.

- $\left\{\begin{matrix}\mathbf{MAX}\\ \mathbf{MIN}\end{matrix}\right\}\left(\left[\left\{\begin{matrix}\mathbf{ALL}\\ \mathbf{DISTINCT}\end{matrix}\right\}\right]\ \mathbf{field}\right)$ oder $\left\{\begin{matrix}\mathbf{MAX}\\ \mathbf{MIN}\end{matrix}\right\}$**([ALL] expression)**

  Gibt das Maximum oder Minimum jeder Gruppe zurück.

Die fünf Funktionen (**AVG, COUNT, SUM, MIN, MAX**) werden Aggregatfuntkionen genannt und diese können in dem Ergebnis als neue Spalten oder in einer Bedingung aus der HAVING Klausel vorkommen. Weil diese Funktionen auf einer Gruppe von Tupel angewendet werden, müssen diese zusammen mit einer GROUP BY Klausel benutzt werden, sonst wird ein einziges Tupel für die ganze Tabelle zurückgegeben (die Tabelle wird als eine Gruppe betrachtet).
Im Allgemeinen, darf eine SELECT Klausel keine Attribute enthalten, die nicht in der GROUP BY Klausel vorkommen außer wenn diese Parameter in einer Aggregatfunktion sind. Falls solche Attribute vorkommen und falls kein Fehler von dem Dantebankmanagement System angezeigt wird, dann wird einen zufälligen Wert aus der Gruppe ausgewählt und in dem Ergebnis angezeigt.
Zwei Tabellen mit derselben Anzahl von Spalten mit denselben Datentypen auf jeder Position (d.h. n-te Spalte in der ersten Tabelle hat denselben Datentyp mit der n-ten Spalte in der zweiten Tabelle) können mit dem **UNION** Operator **vereinigt** werden. Die Tabelle, die zurückgegeben wird, enthält alle möglichen Tupeln (für **ALL**) oder nur die verschiedenen Tupel (ohne **ALL**). Die **ORDER BY** Klausel kann nur in der letzten SELECT Anweisung benutzt werden.
Zwischen zwei Ergebnisse von SELECT Anweisungen kann man INTERSECT oder EXCEPT Operatoren benutzen (oder MINUS).
In einer SELECT Anweisung müssen die Klausen in folgender Reihenfolge benutzt werden:
SELECT expression_list FROM … WHERE … GROUP BY … HAVING … ORDER BY …
Das Ergebnis einer Select Anweisung kann in einer Sicht gespeichert werden:

CREATE VIEW view_name AS SELECT_command

2.4. Aufgaben

I.

a) Erstelle eine relationale Datenbank, in 3NF, die von einer Software Firma benutzt wird um folgende Informationen zu verwalten:

- Aktivitäten: Aktivität Id, Beschreibung, Aktivität Typ
- Angestellte: Angestellte Id, Name, Liste von Aktivitäten, Team (ein Angestellte ist Teil eines Teams) und Team Leader

wobei:

- Eine Aktivität von dem Aktivität Id identifiziert wird
- Ein Angestellte von dem Angestellte Id identifiziert wird

Begründe, dass das erhaltene Datenmodel in 3NF ist.

b) Für das Datenmodel erhalten bei Punkt a, schreibe Anfragen in relationalen Algebra und TSQL um folgende Informationen auszugeben:
   - Gebe die Namen aller Angestellten aus, die wenigstens eine Aktivität mit dem Typ „Design“ und keine Aktivität mit dem Typ „Testing“ haben
   - Gebe die Namen aller Team Leaders aus, welche für ein Team mit wenigstens 10 Angestellte verantwortlich sind

II.

a) Erstelle eine relationale Datenbank, in 3NF, die von der Universität benutzt wird um folgende Informationen zu verwalten:
   - Vorlesungen: Id, Titel, Anzahl ECTS, Liste von Studenten die eine Prüfung für diese Vorlesung geschrieben haben
   - Studenten: Id, Name, Geburtsdatum, Gruppe, Studiumjahr, Studiengang, Liste von Vorlesungen für welche der Student eine Prüfung geschrieben hat zusammen mit dem Datum und die Note bei der Prüfung

   Begründe, dass das erhaltene Datenmodel in 3NF ist.

b) Für das Datenmodel erhalten bei Punkt a, schreibe Anfragen in relationalen Algebra und TSQL um folgende Informationen auszugeben:
   - Gebe die Liste der Studenten (Name, Gruppe, Anzahl der bestandenen Kurse) aus, die in dem Jahr 2013 die Prüfung für mehr als 5 Vorlesungen bestanden haben. Wenn ein Student mehr als eine bestandene Prüfung für eine Vorlesung hat, dann zählt das als ein einziger bestandener Kurs.

III.

a) Erstelle eine relationale Datenbank, in 3NF, die von der Universität benutzt wird um folgende Informationen über Studenten, welche für die Lizenzprüfung angemeldet sind, zu verwalten:
   - CNP, Serie, Id und Name des Studiengangs, Titel der Lizenzarbeit, Id und Name des Betreuers, Id und Name der Abteilung des Betreuers, Liste mit den Software Ressourcen die für den Vortrag gebraucht werden (z.B. .NET C#, C++, usw.), Liste von Hardware Ressourcen die für den Vortrag gebraucht werden (z.B. RAM 8GB, DVD Reader, usw.).

   Begründe, dass das erhaltene Datenmodel in 3NF ist.

b) Für das Datenmodel erhalten bei Punkt a, schreibe Anfragen in relationalen Algebra und TSQL um folgende Informationen auszugeben:
    - Liste der Studenten (Name, Titel der Lizenzarbeit, Name des Betreuers) für welche der Betreuer Teil einer gegebenen Abteilung ist (gegeben wird der Name der Abteilung)
    - Für eine gegebene Abteilung, gebe die Anzahl der Studenten aus, die ein Betreuer aus der entsprechenden Abteilung haben
    - Liste aller Betreuern, die ein Student bei dem Linzenzprojekt betreut haben
    - Liste aller Studente, die folgende zwei Software Ressourcen brauchen: Oracle und C#

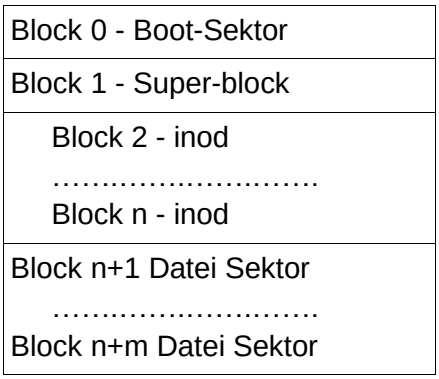
Teil 3. Betriebssysteme

3.1. Dateisysteme Struktur von UNIX

3.1.1. Unix File System. Die interne Struktur des UNIX Plattenlaufwerk

3.1.1.1. Blöcke und Partitionen

Ein UNIX-Dateisystem ist auf einem Datenträger wie einer Festplatte oder einer CD oder auf einer Partition der Festplatte gehostet. Die Festplattenpartitionierung einer Festplatte ist unabhängig vom Betriebssystem, das im Datenträger gehostet wird. Deshalb, sowohl Datenträger als auch physische Systeme werden wir allgemeine UNIX Datenträger nennen.



Datenstruktur für einen Unix-Datenträger

Eine Unix Datei ist eine Folge von Bytes, wobei jedes Byte individuell adressierbar ist. Auf ein Byte kann sowohl nacheinander als auch direkt zugegriffen werden. Der Datenaustausch zwischen dem Speicher und der Festplatte geschieht in Blöcken. Ältere Systeme haben eine Blockgröße von 512 Bytes. Neuere Systeme nutzen größere Blöcke bis zu 4KB für ein effizienteres Platzmanagement. Ein UNIX-Dateisystem ist auf einer Festplatte strukturiert hinterlegt und in vier Blocktypen gegliedert wie es im obigen Bild dargestellt ist.

Block 0 beinhaltet das Betriebssystem bootloader. Dies ist ein Programm abhängig von der Maschinenarchitektur, auf der es installiert ist.

Block 1, der sogenannte Superblock, beinhaltet Informationen, welche das Dateisystemlayout definieren. Solche Informationen sind:

- die Anzahl der i-nodes (wird im nächsten Kapitel erklärt);
- die Anzahl der Festplattenzonen;
- Verweis zur i-node-Zuordnungsliste;
- Verweis zu freiem Speicherplatz;
- Festplattenabmessungen.

Block 2 bis n sind i-nodes, wobei n definiert ist, wenn die Festplatte formatiert ist. Ein i-node ist der Name von UNIX für einen File Descriptor. Die i-nodes sind auf der Festplatte als Liste gespeichert, die sich i-list nennt. Die Seriennummer eines i-nodes in einer i-list besteht aus zwei Bytes und wird i-number genannt. Diese i-number ist der Link zwischen der Datei und dem Anwenderprogramm.

Der größte Teil der Festplatte ist reserviert für Dateien. Die Speicherplatzvergabe für Dateien geschieht durch einen eleganten Indizierungsmechanismus. Der Startpunkt für diesen Vorgang ist im i-node gespeichert.

3.1.1.2 Verzeichnisse und I-Nodes

Das folgende Bild zeigt die Struktur eines Dateieintrags in einem Verzeichnis:

Dateiname (praktisch von unbegrenzter Länge)	i-number

Die Struktur eines Dateieintrags in einem Verzeichnis

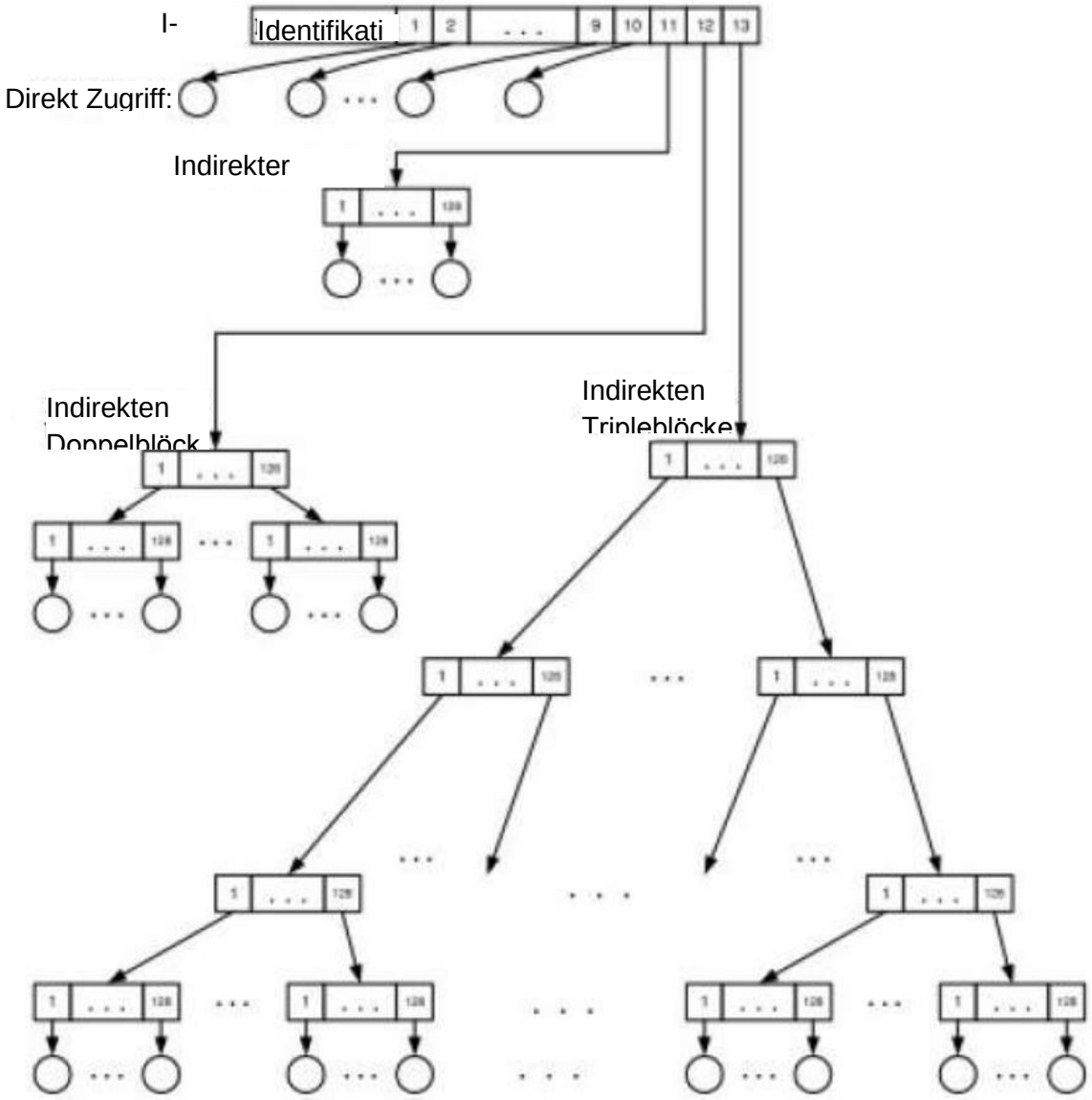
Der Verzeichniseintrag beinhaltet nur die Dateinamen und die dazugehörigen i-node Referenzen. Ein i-node belegt gewöhnlich zwischen 64 und 128 Bytes und beinhaltet in folgender Tabelle aufgeführten Informationen:

mode (modus)	Dateizugriffsrechte
link count (Linkzähler)	Anzahl von Verzeichnissen, welche Referenzen zu dieser i-number beinhalten. Grundsätzliche Anzahl der Links zu dieser Datei
user ID (Benutzer-ID)	Benutzer-ID des Eigentümers
group ID (Gruppen-ID)	Gruppen-ID des Eigentümers
size (Größe)	Die Anzahl der Bytes in der Datei (Dateilänge)
access time (Zugriffszeit)	Zeit des letzten Zugriffs auf die Datei
mod time (Bearbeitungszeit)	Zeit der letzten Bearbeitung der Datei
inode time (i-node-Zeit)	Zeit der letzten Bearbeitung des i-nodes
block list (Blockliste)	Adressliste der ersten Datenblöcke
indorect list (Indirekte Liste)	Bezieht sich auf den Rest der Blöcke, welche zur Datei gehören

3.1.1.2. Verfahren zur Zuordnung (allocation) von Blöcken zu Dateien

Jedes UNIX-Dateisystem besitzt einige individuelle Festwerte wie zum Beispiel: Blockgröße, i-node-Größe, Plattenadressgröße, wie viele initiale Blockadressen direkt in der i-node und wie viele Referenzen in der indirekten Liste gespeichert werden. Unabhängig von diesen Werten sind die Speicher- und Abrufprinzipien gleich.

Um dies zu veranschaulichen werden wir Werte benutzen, die häufig im verbreiteten Dateisystem gefunden werden. Es wird ein 512 Byte großer Block betrachtet, die Plattenadresse ist 4 Bytes groß. Demnach können in einem Block 128 solcher Adressen gespeichert werden. Es wird weiterhin berücksichtigt, dass das i-node die Adressen der ersten 10 Blöcke der Datei speichert und die indirekte Liste 3 Elemente enthält. Mit diesen Werten zeigt das untenstehende Bild wie die i-node auf die Datenblöcke verweist.



Struktur eines INode und Zugriff auf der Datei Blöcker

In der i-node existiert eine Liste bestehend aus 13 Einträgen, wobei jeder Eintrag zu den physischen Blocks der Datei verweist.
Die ersten 10 Einträge beinhalten die Adressen der ersten 10 512-Byte-Block der Datei;
Eintrag 11 besteht aus der Adresse mit dem Namen: indirekter Block. Dieser Block enthält die Adressen von den nächsten 128 512-Byte-Blocks der Datei;
Eintrag 12 beinhaltet die Adresse von eines Blocks namens indirekter Doppelblock. Dieser Block enthält die Adressen von 128 indirekten Blöcken. Jeder dieser Blöcke besteht aus 128 512-Byte-Blöcken;
Eintrag 13 beinhaltet die Adresse von eines Blocks namens indirekter Tripleblock. Dieser Block enthält die Adressen von 128 indirekten Doppelblöcken. Jeder dieser indirekten Doppelblöcke besteht aus 128 indirekten Blöcken, welche nochmals die Adressen von 128 512-Byte-Blöcken beinhalten.

In der obigen Abbildung wurden Kreise verwendet um Dateiblöcke zu symbolisieren und Rechtecke für Referenzblöcke. Aus der Abbildung lässt sich einfach herauslesen, dass die maximalen Festplattenzugriffe für einen Teil der Datei höchstens vier beträgt. Für kleine Dateien ist die Anzahl geringer. Solange die Datei geöffnet ist, ist ihre i-node geladen und im geräteeigenen Speicher vorhanden. Die folgende Tabelle zeigt die Anzahl der erforderlichen Festplattenzugriffe um je nach Dateilänge den direkten Zugriff auf jedes Dateibyte zu erhalten.

Maxime Länge (Blöcke)	Maxime Länge (Bytes)	Indirekter Zugriff	Zugriff auf Informationen	Gesamt-zugriffe
10	5120	-	1	1
10+128 = 138	70656	1	1	2
10+128+128^2 = 16522	8459264	2	1	3
10+128+128^2+128^3 = 2113674	1082201088	3	1	4

Neuere UNIX-Versionen nutzen Blöcke bestehend aus 4096 Bytes, welche 1024 Referenzadressen speichern können. Die i-node speichert 12 direkte Zugriffsadressen. Unter diesen Bedingungen verändert sich die obige Tabelle wie folgt:

Maxime Länge (Blöcke)	Maxime Länge (Bytes)	Indirekter Zugriff	Zugriff auf Informationen	Gesamt-zugriffe
12	49152	-	1	1
12+1024 = 1036	4243456	1	1	2
12+1024+1024^2 = 1049612	4299210752	2	1	3
12+1024+1024^2+1024^3 = 1073741824	4398046511104 (über 5000Go)	3	1	4

3.1.2. Dateitypen und -systeme

Das UNIX-Dateisystem kann acht Dateitypen aufrufen:

1. Normale Datei (regular file)
2. Verzeichnisse (directory)
3. Hardlinks (hard link)
4. Symbolische Links (symbolic link)
5. Netzwerk-Kommunikationsendpunkte (socket)
6. Feststehende Programmverbindungen (named pipe)
7. Zeichenorientierte Geräte (char device)
8. Blockorientierte Geräte (block device)

Neben diesen acht Typen gibt es vier weitere Einheiten, die das UNIX-System syntaktisch genau so behandelt als wären diese Dateien. Diese Entitäten sind für Kommunikation zwischen Prozesse bei Unix Kern verwaltet und sie existiert in Kern physisch.

9. Pipes (anonymous pipes)
10. Shared memory segments
11. Message queues
12. Semaphore (Semaphores)

Normale Dateien sind Sequenzen aus Bytes, die entweder sequentiell oder direkt durch die Nutzung der Seriennummer zugänglich sind.

Verzeichnisse. Eine Verzeichnisdatei unterscheidet sich von einer normalen Datei nur durch die Informationen, die sie enthält. Ein Verzeichnis beinhaltet die Liste der Namen und Adressen von den Dateien, die es enthält. Normalerweise hat jeder Nutzer sein eigenes Verzeichnis, welches auf normale Dateien und Unterverzeichnisse verweist.

Spezielle Dateien. Zu dieser Kategorie gehören bisher die letzten sechs Dateitypen. Tatsächlich betrachtet UNIX jedes Eingabe- und Ausgabegerät als spezielle Datei. Aus Nutzersicht gibt es keinen Unterschied zwischen der Arbeitsweise mit einer normalen Datei und der mit einer speziellen Datei.

Jedes Verzeichnis umfasst diese zwei speziellen Eingaben:
“.” (Punkt) verweist auf das Verzeichnis selbst
“..” (zwei aufeinanderfolgende Punkte) verweist auf das übergeordnete Verzeichnis

Jedes Dateisystem besitzt ein Hauptverzeichnis mit dem Namen root oder /.

Normalerweise verwendet jeder Benutzer ein aktuelles Verzeichnis, das ihm nach Systemeintritt zugeordnet ist. Der Nutzer kann das Verzeichnis wechseln (cd), ein neues Verzeichnis im aktuellen Verzeichnis erstellen (mkdir), ein Verzeichnis löschen (rmdir), den Zugriffspfad vom Hauptverzeichnis aus anzeigen lassen (pwd).

Das Auftreten einer hohen Anzahl von UNIX Verteilern/Händlern führt zwangsläufig zum Auftreten von vielen geschützten “erweiterten Dateisystemen”. Beispielsweise:
Solaris nutzt das UFS Dateisystem
Linux nutzt hauptsächlich ext2 und neuerdings ext3
IRIX nutzt XFS
Etc.

Die heutigen UNIX-Distributionen unterstützen auch Dateisysteme, die zu anderen Betriebssystemen gehören. Die wichtigsten dieser Dateisysteme sind weiter unten aufgelistet. Die Verwendung dieser Dritt-Dateisysteme ist für den Benutzer durchschaubar. Jedoch wird dem User empfohlen, dieses System sehr sorgfältig für alle anderen Funktionen als das Lesen zu verwenden. Wird ein Microsoft Word Dokument in UNIX bearbeitet, ist das Ergebnis unbrauchbar.
FAT und FAT32 ist speziell für MS-DOS und Windows 9x
NTFS ist speziell für Windows NT und 2000

UNIX-Systemadministratoren müssen auf die verwendeten Dateitypen und die Berechtigungen achten, die sie den Benutzern zuteilen.

Das Baumstrukturprinzip eines Dateisystems besagt, dass jede Datei oder jedes Verzeichnis ein einzelnes Elternteil hat. Jede Datei oder jedes Verzeichnis hat einen einzigen eindeutigen Zugriffspfad, der ab dem Hauptverzeichnis root startet. Der Link zwischen einer Datei oder einem Verzeichnis und ihrem übergeordneten Verzeichnis wird als natürlicher Link bezeichnet. Die natürlichen Links werden still erstellt, wenn die Datei oder das Unterverzeichnis erstellt wird.

3.1.2.1 Harte und symbolische Links

Es gibt Situationen in denen es sinnvoll ist einen Teil des Verzeichnisses unter verschiedenen Nutzern zu teilen. Aus diesem Grund muss zum Beispiel eine Datenbank, die irgendwo im Dateisystem gespeichert ist, für viele Benutzer zugänglich sein. UNIX ermöglicht eine solche Funktionsweise mit zusätzlichen Links. Ein zusätzlicher Link erlaubt es, eine Datei auf unterschiedliche Weise zu verweisen als die natürlichen Links. Diese zusätzlichen Links sind: harte Links (harte Verknüpfungen) und symbolische Links (weiche Verknüpfungen).

Harte Links sind identisch zu den natürlichen Links und können ausschließlich vom Systemadministrator erstellt werden. Dieser harte Link ist ein Eintrag im Verzeichnis, der auf die gleiche Unterstruktur verweist wie auch natürliche Links verweisen. Folglich erhält durch den harten Link die Unterstruktur zwei übergeordnete Verzeichnisse. Grundsätzlich wird ein und derselben Datei durch den harten Link ein zweiter Name gegeben. Das ist der Grund, warum bei der Analyse eines Verzeichnisbaums Dateien mit hartem Link doppelt erscheinen. Jedes Duplikat erscheint dabei mit der Anzahl der harten Links, die auf die Datei hinweisen.

Zum Beispiel:
Wenn es eine Datei mit dem Namen “old” gibt und der Systemadministrator den Befehl

```
# ln old newlink
```

ausführt, dann erscheint das Dateisystem mit zwei identischen Dateien: old und newlink, wobei auf beide mit jeweils zwei Links verweisen wird.

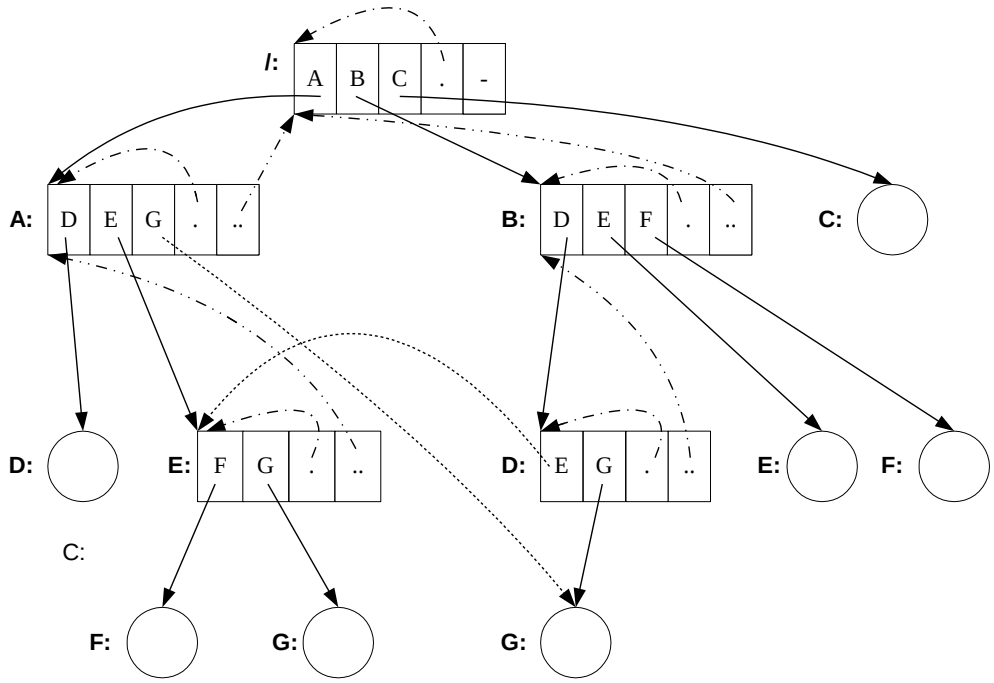
Harte Links können nur im selben Dateisystem erstellt werden (wir geben die Details dazu später).

Die symbolischen Links sind spezielle Einträge in einem Verzeichnis. Sie verweisen auf eine Datei oder auf ein Verzeichnis in der Verzeichnisstruktur. Der Eintrag selbst verhält sich wie ein Unterverzeichnis des Verzeichnisses, in dem der Eintrag erstellt wurde.

In der einfachsten Form wird dabei eine symbolische Verknüpfung mit folgendem Befehl erstellt:

#ln –s pathInTheDirectoryStructure symbolicName

Nachdem der Befehl oben ausgeführt wurde, wird pathInTheDirectoryStructure mit einem zusätzlichen Link gekennzeichnet, und symbolicName verweist (ausschließlich) auf diesen Pfad. Symbolische Links können von üblichen Benutzern verwendet werden (nicht nur vom Systemadministrator wie die harten Links) und sie können auch auf Pfade verweisen die sich außerhalb des Dateisystems befinden.



Die symbolischen und harten Links verwandeln die Baumstruktur des Dateisystems in eine azyklische Graphenstruktur. Das Beispiel oben zeigt eine einfache Verzeichnisstruktur. Großbuchstaben A, B, C, D, E, F und G sind Namen von herkömmlichen Dateien, Verzeichnissen und Links. Es ist möglich, dass derselbe Name mehrfach in der Verzeichnisstruktur erscheint. Dank der hierarchischen Struktur der Verzeichnisse ist es allerdings möglich Unklarheiten zu vermeiden. Normale Dateien werden mit Kreisen und Verzeichnisse mit Rechtecken markiert.

Die Links werden mit drei Pfeiltypen gekennzeichnet:
Ununterbrochene Linie - natürliche Verbindungen
Gestrichelte Linie - Verknüpfung zum selben Verzeichnis oder zum übergeordneten Verzeichnis
Gepunktete Linie - Symbolische und harte Verknüpfungen

Es gibt 12 Knoten im obigen Beispiel mit beiden Verzeichnissen und Dateien. Betrachten wir den Baum a s a ausschließlich unter Berücksichtigung der natürlichen Links gibt es 7 Zweige und 4 Ebenen.

Nehmen wir an, dass die beiden Verbindungen (gestrichelte Linien) symbolisch sind. Um diese Links zu erstellen, müsste man die folgenden Befehle ausführen:

```
cd  /A
ln  -s  /A/B/D/G  G          Erster link
cd  /A/B/D
ln  -s  /A/E  E              Zweiter link
```

Nehmen wir an, dass das aktuelle Verzeichnis B ist. Wir werden den Baum in der Reihenfolge: Verzeichnis, gefolgt von seinen Unterverzeichnissen von links nach rechts analysieren. Wenn zutreffend, werden wir auf die gleiche Zeile mehrere Spezifikationen des gleichen Knotens setzen. Die Einträge, die Links sind, sind unterstrichen. Die längsten 7 Zweige sind mit # rechts markiert.

/	..				
/A	../A				
/A/D	../A/D				#
/A/E	../A/E	D/E	./D/E		
/A/E/F	../A/E/F	D/E/F	./D/E/F		#
/A/E/G	../A/E/G	D/E/G	./D/E/G		#
/B	.				
/B/D	D	./D			
/B/D/G	D/G	./D/G	/A/G	../A/G	#
/B/E	E	./E			#
/B/F	F	./F			#
/C	../C				#

Was passiert beim Löschen mehrerer Links? Was passiert beispielsweise, wenn beispielsweise einer der unten stehenden Befehle ausgeführt wird?

```
rm  D/G
rm  /A/G
```

Es ist klar, dass die Datei nicht betroffen sein muss, wenn sie nur durch eine der Spezifikationen gelöscht wird.

Dazu enthält der Dateideskriptor ein Feld namens link counter. Dieses hat den Wert 1 wenn die Datei erstellt wird und wird bei jedem neuen Link um 1 erhöht. Wenn sie gelöscht wird, wird der Zähler um 1 verringert. Nur wenn er null ist, wird die Datei von der Festplatte gelöscht und die belegten Blöcke werden frei.

3.2. UNIX Prozesse

Unix-Prozesse: erstellen, fork, exec, exit, wait; Kommunikation durch pipe und FIFO

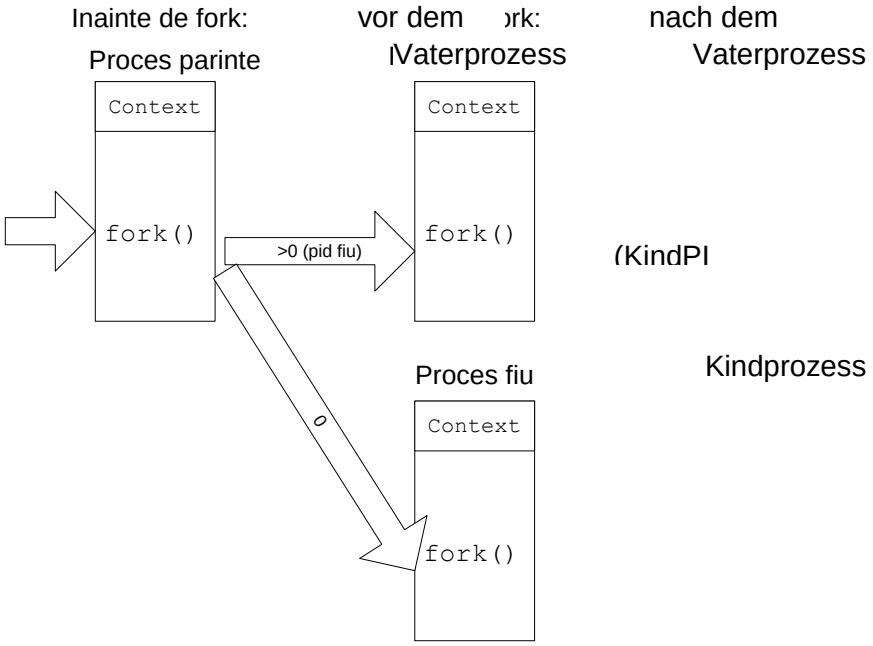
3.2.1. Hauptsystemaufrufe für Prozessverwaltung

In den Abschnitten dieses Kapitels präsentieren wir die wichtigsten Aufrufe für das Arbeiten mit System-Prozessen: fork, exit, wait und exec*. Beginnen wir mit fork(), dem Aufruf zur Erstellung eines Prozesses.

3.2.1.1. Prozesserstellung. Der fork-Systemaufruf

Im UNIX-Betriebssystem wird ein Prozess mit dem fork() Systemaufruf erstellt. Bei normaler Funktionsweise ist der Effekt des Aufrufs folgendermaßen: Das übergeordnete Prozessspeicherbild wird in einen freien Speicherbereich kopiert, diese Kopie ist der neu erstellte Prozess, der in der ersten Phase mit dem anfänglichen Prozess identisch ist. Die beiden Prozesse setzen ihre Ausführung gleichzeitig fort mit der Anweisung nach dem fork() - Aufruf.

Der neu erstellte Prozess wird als Kindprozess bezeichnet und der Prozess, der fork() aufgerufen hat, heißt übergeordneter Prozess. Mit Ausnahme der verschiedenen Adressräume unterscheidet sich der untergeordnete Prozess von dem übergeordneten Prozess nur über seine Prozesskennung (PID), seinen übergeordneten Prozessbezeichner (PPID) und den vom fork() - Aufruf zurückgegebenen Wert. Bei normaler Funktionsweise kehrt fork() in den übergeordneten Prozess (den Prozess, der fork() aufgerufen hat) PID des neuen Kindprozesses. Dort gibt er 0 zurück.



Figur 3.1 Der fork() - Mechanismus

Die obige Abbildung zeigt den fork() - Mechanismus, wobei die Pfeile den Befehl angeben, der gerade im Prozess ausgeführt wird.

Im Falle des Scheiterns gibt fork den Wert -1 und setzt entsprechend die errno Umgebungsvariable. Der fork-Systemaufruf kann auftreten, wenn:
Es ist nicht genügend Arbeitsspeicher zum Erstellen einer Kopie des Elternprozesses vorhanden
Die Gesamtanzahl der Prozesse überschreitet den zulässigen Grenzwert

Dieses Verhalten des fork() -Systemaufrufs macht es einfach, zwei Sequenzen von Anweisungen zu implementieren, die parallel ausgeführt werden:

```
if ( fork() == 0 ) {
        /* Kindprozess Befehle */
}
else {
        /* Vaterprozess Befehle */
}
```

Das folgende Programm veranschaulicht die Verwendung des fork()-Aufrufs:

```
main(){
  int pid,i;
  printf("\nProgrammstart:\n");
  if ((pid=fork())<0) err_sys("Kann nicht fork() ausführen\n");
  else if (pid==0){//Das ist  KINDProzess
    for (i=1;i<=10;i++){
      sleep(2);          //schlaf 2 Sekunden
      printf("\KIND(%d) von VATER(%d): 3*%d=%d\n",
                    getpid(),getppid(),i,3*i);
    }
    printf("Ende von KIND\n");
  }
  else if (pid>0){//Das ist  VATERProzess
    printf("Erstellen KIND(%d)\n",pid);
    for (i=1;i<=10;i++){
      sleep(1);          //schlafe 1 Sekunde
      printf("VATER(%d):2*%d=%d\n",getpid(),i,2*i);
    }
    printf("Ende von VATER\n");
  }
}
```

Wir haben es absichtlich so gemacht, dass der Kindprozess länger als das Elternteil wartet (bei komplexen Berechnungen ist es oft so, dass die Operationen eines Prozesses länger dauern als die des anderen). Folglich beendet das Elternteil die Ausführung früher. Die Ergebnisse sind:

```
Programmstart:
Erstellen KIND(20429)
VATER(20428): 2*1=2
        KIND(20429) von VATER(20428): 3*1=3
VATER(20428): 2*2=4
VATER(20428): 2*3=6
        FIUL(20429) von VATER(20428): 3*2=6
VATER(20428): 2*4=8
VATER(20428): 2*5=10
```

KIND(20429) von VATER(20428): 3*3=9
VATER(20428): 2*6=12
VATER(20428): 2*7=14
KIND(20429) von VATER(20428): 3*4=12
VATER(20428): 2*8=16
VATER(20428): 2*9=18
KIND(20429) von VATER(20428): 3*5=15
VATER(20428): 2*10=20
Ende von VATER
KIND(20429) von VATER(1): 3*6=18
KIND(20429) von VATER(1): 3*7=21
KIND(20429) von VATER(1): 3*8=24
KIND(20429) von VATER(1): 3*9=27
KIND(20429) von VATER(1): 3*10=30
Ende von KIND

3.2.1.2 Ausführen eines externen Programms. Die exec-Systemaufrufe

Fast alle Betriebssysteme und Programmierumgebungen bieten auf die eine oder andere Weise Mechanismen für die Ausführung eines Programms aus einem anderen Programm. UNIX bietet diesen Mechanismus mit dem exec* Systemaufruf an. Wie wir sehen werden, bietet die kombinierte Nutzung von fork und exec eine große Elastizität bei der Abwicklung von Prozessen.

Die Sytemaufrufe der exec-Familie starten ein neues Programm im selben Prozess. Der Exec-Systemaufruf erhält den Namen einer ausführbaren Datei und der Inhalt dieser Datei überschreibt den vorhandenen aktuellen Prozess, wie in Abbildung 3.2 unten gezeigt.

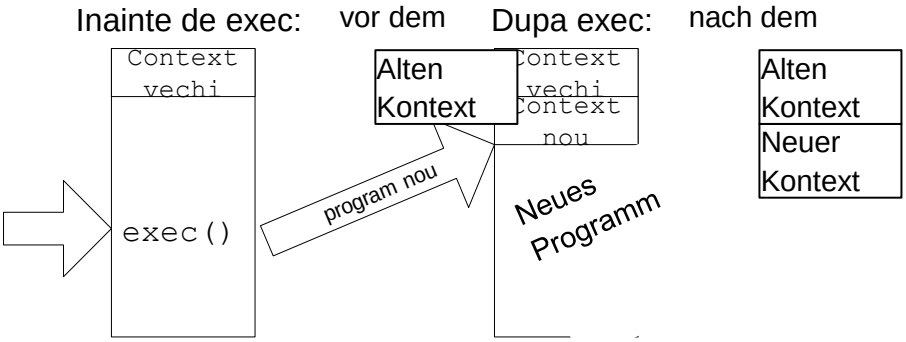


Abb. 3.1 Der exec - Mechanismus

Nach dem Aufruf von exec werden die Anweisungen im aktuellen Programm nicht mehr ausgeführt. An deren Stelle werden die Anweisungen des neuen Programms ausgeführt.

UNIX bietet sechs Exec-Systemaufrufe nach drei Kriterien an:
Der Pfad zur ausführbaren Datei: absolut oder relativ
Umgebungsvererbung oder Schaffung einer neuen Umgebung für das neue Programm
Befehlszeilenargumente Spezifikation: spezifische Liste oder Verweisfeld

Von den acht möglichen Kombinationen, die durch die drei oben genannten Kriterien erzeugt werden, werden die beiden mit relativem Pfad und neuer Umgebung eliminiert. Die Prototypen der sechs exec-Funktionsaufrufe sind:

```
int execv (char *datei, char *argv[]);
int execl (char *datei, char *arg0, …, char *argn, NULL);
int execve(char *datei, char *argv[], char *envp[]);
int execle(char *datei, char *arg0, …, char *argn, NULL,
        char *envp[]);
int execvp(char *datei, char *argv[]);
int execlp(char *datei, char *arg0, …, char *argn, NULL);
```

Die Bedeutung der Parameter ist wie folgt:

datei - der Name der ausführbaren Datei, die das aktuelle Programm ersetzen wird. Er muss mit dem Argument argv[0]oder arg0 übereinstimmen. Argv ist Dashboard

argv - eine Reihe von Verweisen, die in NULL enden, die die Befehlszeilenargumente für das neue Programm enthält

arg0, arg1, ... , argn, NULL sind die Befehlszeilenargumente des auszuführenden Programms - explizit als Strings angegeben. Der letzte muss auch NULL sein.

envp Array von Zeigern, endet ebenfalls in NULL, das eine Zeichenfolge enthält, die den neuen Umgebungsvariablen entspricht, im Format "name = value"

3.2.1.2. Systemaufrufe exit und wait

Der Systemaufruf: exit(int n) bewirkt, dass der aktuelle Prozess beendet wird und die Rückkehr auf den Elternprozess (der über fork erstellt wurde). Integer n ist der Exit-Code des Prozesses. Wenn der übergeordnete Prozess nicht mehr existiert, wird der Prozess in einen Zombie-Zustand verschoben und mit dem init-Prozess (PID 1) verknüpft.

Erwarten das Ende eines Prozess wird durchgeführt unter Verwendung von einem der Systemaufrufen wait() oder waitpid(). Ihre Prototypen sind:

```
pid_t wait(int *Kondition)
pid_t waitpid(pid_t pid, int *Kondition, int Optionen);
```

Die Systemaufrufe wait() werden verwendet um auf die Beendigung eines Prozesses zu warten. Der Aufruf wait() unterbricht die Ausführung des aktuellen Prozesses bis ein untergeordneter Prozess endet. Wenn der untergeordnete Prozess vor dem wait() Aufruf beendet wurde, endet der Aufruf sofort. Nach Beendigung des wait() Aufrufs sind die Ressourcen des untergeordneten Prozesses frei.

3.2.2. Kommunikation zwischen Prozessen über pipe

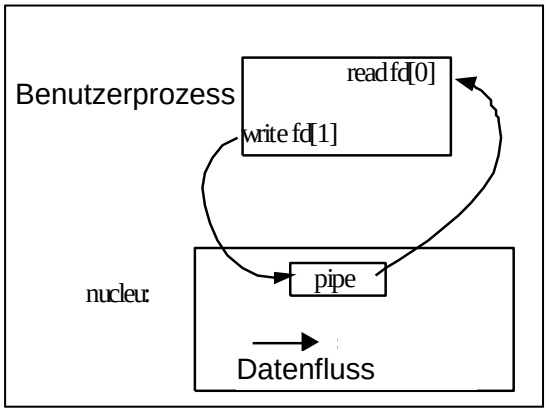
3.2.2.1 Das Konzept von pipe

Das PIPE-Konzept erschien zum ersten Mal im UNIX-System, um einem Kindprozess die Kommunikation mit seinem übergeordneten Prozess zu erlauben. Normalerweise leitet der übergeordnete Prozess seine Standardausgabe (stdout) in Richtung pipe und der Kindprozess leitet seine Standardeingabe (stdin) aus Richtung pipe. Die meisten Betriebssysteme verwenden den Operator "|", um diese Art der Verbindung zwischen den Betriebssystembefehlen zu markieren.

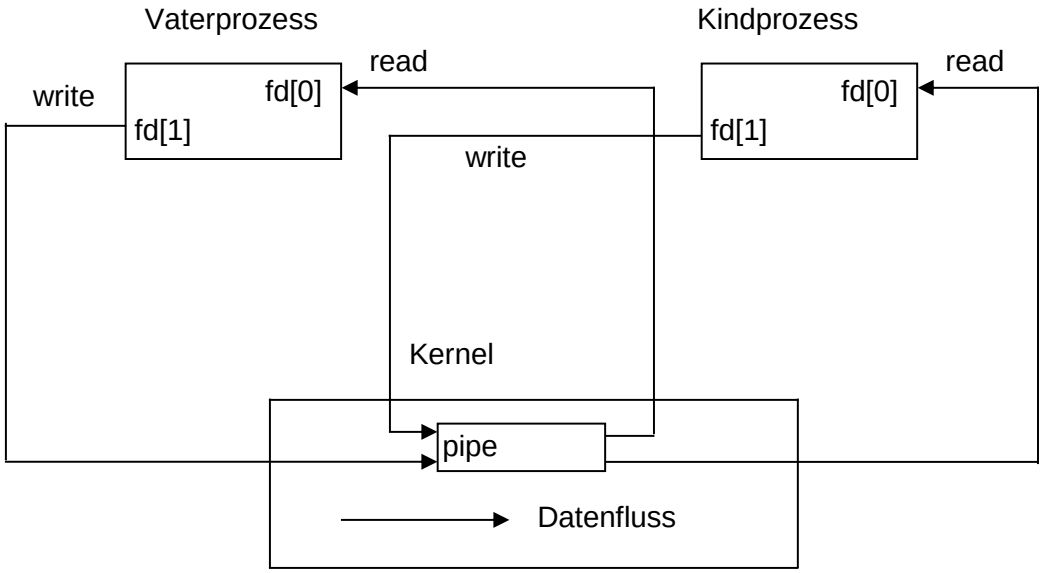
Ein Unix-Pipe ist ein One-Way-Datenfluss vom Kernel verwaltet. Grundsätzlich weist der Kernel einen Puffer (buffer) von 4096 Bytes auf, der wie oben beschrieben verwaltet wird. Die Erstellung einer Pipe erfolgt über den folgenden Systemaufruf:

int pipe(int fd[2]);

Der Integer fd[0] ist ein Dateideskriptor, der verwendet werden kann, um aus der Pipe zu lesen, und fd[1] ist ein Dateideskriptor, der das Schreiben in die Pipe erlaubt. Nach der Erstellung erscheint die Verknüpfung zwischen dem Benutzer und dem von dieser Pipe erzeugten Kernel wie im Bild unten.



Ein Pipe macht in einem einzigen Prozess offensichtlich nicht viel Sinn. Es ist wichtig, dass pipe und fork zusammenarbeiten. Folglich wird nach dem Erstellen der pipe fork aufgerufen. Die Verbindung zwischen den beiden Prozessen mit dem Kernel sieht wie folgt aus:



Die One-Way-Kommunikation durch die Pipe bleibt ganz vom Entwickler gewährleistet. Um das zu sichern sollte der Entwickler die unnötigen Pipes beenden, bevor die Kommunikation gestartet wird.
Im übergeordneten Prozess aufrufen close(fd[0]);
Im Kindprozess aufrufen close(fd[1]);

Wenn die gewünschte Richtung der Kommunikation umgekehrt wird, dann sollten die oben genannten Operationen auch umgekehrt werden: das Elternteil führt close(fd[1]) und das Kind close(fd[0] aus.

3.2.2.2. Beispiel: Umsetzung von who|sort durch pipe und exec

Betrachten wir nun den shell-Aufruf bestehend aus:

$ who | sort

Wir zeigen, wie die Verbindung zwischen den beiden Befehlen durch Pipe erzeugt wird. Der übergeordnete Prozess (der den shell-Prozess ersetzt) erzeugt zwei untergeordnete Prozesse. Diese beiden leiten ihre Ein- und Ausgänge entsprechend um. Der erste Kind-Prozess führt den Befehl who aus und der zweite Kind-Prozess führt den Befehl sort aus, während der Elternprozess auf seine Fertigstellung wartet. Der Quellcode wird im folgenden Programm dargestellt.

```
//whoSort.c
//Starten in pipe Shell-Befehle: $ who | sort
#include <unistd.h>

main (){

    int p[2];
```

```
    pipe (p);
    if (fork () == 0) {             // erster Sohn
      dup2 (p[1], 1);               //umlenken Standardausgabe
      close (p[0]);
      execlp ("who", "who", 0);
    }
    else if (fork () == 0) {        // zweiter Sohn
      dup2 (p[0], 0);               // umlenken Standardeingabe
      close (p[1]);
      execlp ("sort", "sort", 0);// sort Ausführung
    }
    else {                                  // Vater
      close (p[0]);
      close (p[1]);
      wait (0);
      wait (0);
    }
}
```

Hinweis: Um den oben genannten Code besser zu verstehen, sollte der Leser den Systemaufruf dup2 aus dem UNIX-Handbuch verstehen. Hier nimmt dup2 als Parameter den Pipe-Deskriptor an.

3.2.3. Kommunikation zwischen Prozessen über FIFO

3.2.1.1. Das Konzept von FIFO

Der Hauptnachteil der Verwendung von Pipe in UNIX ist, dass es nur bei verwandten Prozessen angewendet werden kann: Die Prozesse, die durch Pipe kommunizieren, müssen Nachkommen des Prozesses sein, der das Pipe erzeugt. Dies ist notwendig, damit die Pipe-Deskriptoren von dem mit fork erzeugten Kinderprozessen vererbt werden.

UNIX System V (um 1985) führte das Konzept von FIFO (genannt Pipes) ein. Dies ist ein One-Way-Datenfluss, auf den durch eine auf dem Datenträger gespeicherte Datei im Dateisystem zugegriffen wird. Der Unterschied zwischen Pipe und FIFO ist, dass FIFO einen Namen hat und im Dateisystem gespeichert ist. Aus diesem Grund kann auf einen FIFO von allen Prozessen zugegriffen werden, die kein gemeinsames Elternteil haben. Auch wenn das FIFO auf der Festplatte gespeichert ist, sind keine Daten auf der Festplatte gespeichert. Die Daten werden in Systemkernelpuffern verarbeitet.

Vom Konzept her ähneln sich FIFO und Pipe. Die wesentlichen Unterschiede zwischen ihnen sind die folgenden:
Pipe wird im RAM-Speicher verwaltet, der vom Kernel verwaltet wird, während dies für FIFO auf dem Datenträger erfolgt
Alle Prozesse, die durch Pipe kommunizieren, müssen Nachkommen des Erstellungsprozesses sein, während das für FIFO nicht notwendig ist

Die Erstellung eines FIFO erfolgt mit einem der folgenden Systemaufrufe:

int mknod (char *FIFO_Name, int mode, 0);
int mkfifo (char *FIFO_Name, int mode);

oder unter Verwendung der folgenden Shell-Befehle:

$ mknod numeFIFO p
$ mkfifo numeFIFO

Der Namensparameter ist der Name der Datei vom Typ FIFO
Das Argument "mode" gibt die Zugriffsberechtigungen für die FIFO-Datei. Bei der Verwendung von mknod muss der Modus neben der Zugriffsberechtigung das Flag S_IFIFO angeben (verbunden durch den bitwise-OR-Operator). Dieses Flag ist in <sys / stat.h> definiert
Der letzte Parameter des mknod Systemaufrufs wird ignoriert und deshalb haben wir dort eine 0.
Bei der Verwendung des Befehls mknod muss der letzte Parameter "p" sein, um dem Befehl mitzuteilen, dass eine Pipe erstellt wird

Die beiden oben genannten Funktionsaufrufe, obwohl von POSIX angegeben, sind beide keine Systemaufrufe in allen UNIX-Implementierungen. FreeBSD implementiert sie beide als Systemaufrufe, aber bei Linux und Solaris ist nur mknod ein Systemaufruf, während mkfifo eine mit mkfifo implementierte Bibliotheksfunktion ist. Die beiden shell-Befehle sind auf den meisten UNIX-Implementierungen verfügbar. Bei älteren UNIX-Distributionen sind die Befehle nur für den Superuser zugänglich, aber ab UNIX System 4.3 stehen sie auch für reguläre Benutzer zur Verfügung.

Um einen FIFO zu löschen kann man entweder den Befehl rm oder den C-Funktionsaufruf unlink() verwenden.

Sobald die FIFO erstellt ist, muss es zum Lesen und Schreiben mit dem Systemaufruf open geöffnet werden. Die Funktionsweise, seine Effekte und die Effekte des O_NDELAY-Flags, die dem open Systemaufruf gegeben werden, sind in der folgenden Tabelle dargestellt.

Operation	ohne die O_NDELAY Flagge	mit dem O_NDELAY Flagge
FIFO read-only, es gibt keinen Prozess, der es zum Schreiben öffnet	Wartet, bis es einen Prozess gibt, der es zum Schreiben öffnet	wird sofort beendet, ohne Fehlermeldung
FIFO write-only, es gibt keinen Prozess, der es zum Lesen öffnet	Wartet, bis es einen Prozess gibt, der es zum Lesen öffnet	wird sofort mit Fehlermeldung zurückgegeben: errno to ENXIO
Lesen von FIFO oder Pipe, wenn keine Daten	Wartet, bis es Daten in Pipe oder FIFO gibt oder bis es keinen	wird sofort mit Wert 0 zurückgegeben

vorhanden sind	Prozess gibt, der es zum Schreiben öffnet. Gibt Länge der gelesenen Daten zurück oder 0, wenn kein Prozess zum Schreiben mehr vorhanden ist.	
Schreibt in FIFO oder Pipe, wenn sie voll sind	Wartet, bis Platz zum Schreiben vorhanden ist und schreibt so viel wie möglich	wird sofort mit Wert 0 zurückgegeben

3.2.3.2. Beispiel: Client/Server-Kommunikation durch FIFO

Das Client/Server-Anwendungsmodell ist klassisch. Im Folgenden werden wir einen Umriss der Client/Server-Applikation präsentieren, die über das FIFO kommuniziert. Um sicherzustellen, dass die Kommunikation in zwei Richtungen verläuft, werden wir zwei FIFOs nutzen. Die spezifische Logik der Applikation ist nicht gegeben und wird durch Funktionsaufrufe an client(int in, int out) und server(int in, int out) ersetzt. Jeder von ihnen bekommt die Dateideskriptoren als Parameter, durch welche sie mit ihrem Partner kommunizieren.

Die beiden Programme verlassen sich auf die FIFOs, die vorher mit shell-Befehlen erstellt und anschließend mit shell-Befehlen gelöscht werden.

Server-Programm:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <stdio.h>
#include <unistd.h>

#include "server.c"

#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"

main() {
  int    readfd, writefd;

 - - - - - - - - - - - - - -

  readfd = open (FIFO1, 0));

  writefd = open (FIFO2, 1));

  for ( ; ; ) {  // Endlosschleife für Warteanforderungen
```

```
        server(readfd, writefd);
    }

 - - - - - - - - - - - - - -

    close (readfd);
    close (writefd);
}
```

Client-Programm:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <stdio.h>
#include <unistd.h>

#include "client.c"

extern int errno;

#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"

main() {
    int    readfd, writefd;

 - - - - - - - - - - - - - -

    writefd = open (FIFO1, 1));

    if ((readfd = open (FIFO2, 0));

    client(readfd, writefd);

 - - - - - - - - - - - - -

    close (readfd);
    close (writefd);
}
```

3.3. Interpreter von Batch-Dateien

3.3.1. Funktionsweise des Shell-Befehls-Interpreter

Ein Befehlsinterpreter (Shell) ist ein spezielles Programm, das eine Schnittstelle zwischen dem Benutzer und dem UNIX-Betriebssystem (Kernel) zur Verfügung stellt. Aus dieser Perspektive kann der Shell-Interpreter auf zwei Arten betrachtet werden:

Befehlssprache, die als Schnittstelle zwischen dem Benutzer und dem System fungiert. Wenn ein Benutzer eine Arbeitssitzung öffnet, wird ein Shell implizit gestartet und fungiert als Befehlsinterpreter. Der Shell-Befehl zeigt dem Standardausgang (in der Regel ein Terminal) eine Eingabeaufforderung an und bietet dem Benutzer Mittel zum Ausführen von Befehlen.
Programmiersprache, die als Basiselemente die UNIX-Befehle nutzt (semantisch ähnlich der Zuordnungsanweisung in anderen Programmiersprachen). Das primitive Element für die Bedingungen ist der Exit-Code des letzten Befehls. Zero bedeutet TRUE und jeder andere Wert bedeutet FALSE. Shells haben die Begriffe von Variablen, Konstanten, Ausdruck, Kontrollstruktur und Unterprogramm. Die von Shell verwendeten Ausdrücke sind meist Zeichenfolgen. Die syntaktischen Aspekte wurden auf ein Minimum reduziert.

Mit dem Start bleibt eine Shell aktiv, bis sie spezifisch geschlossen wird und folgt dem folgenden Algorithmus:

```
Während (Sitzung ist nicht geschlossen)
        Anzeigeaufforderung;
        Befehlszeile lesen;
        Wenn (Kommandozeile endet mit '&') dann
                Erstellen Sie einen Prozess, der den Befehl ausführt
                Warten Sie nicht, bis die Ausführung abgeschlossen ist
        Sonst
                Erstellen Sie einen Prozess, der den Befehl ausführt
                Warten Sie, bis die Ausführung abgeschlossen ist
        Ende Wenn
Ende, während
```

Der oben beschriebene Algorithmus unterstreicht zwei Modi, in denen ein Befehl ausgeführt werden kann:
Modus Vordergrund - sichtbare Ausführung. Shell wartet darauf, dass die Ausführung beendet wird und eine Aufforderung erneut angezeigt wird. Dies ist der implizite Ausführungsmodus für jeden UNIX-Befehl.
Modus Hintergrund - versteckte Ausführung. Shell hat den Befehl ausgeführt, aber wartet nicht darauf, dass es fertig ist und zeigt die Eingabeaufforderung sofort nach dem Start an, um dem Benutzer die Möglichkeit zu bieten, einen weiteren Befehl auszuführen. Um einen Befehl im Hintergrund laufen zu lassen, muss man ihn mit '&' beenden.

In einer UNIX-Sitzung kann man eine beliebige Anzahl von Hintergrundbefehlen ausführen, aber nur einen Vordergrundbefehl. Beispielsweise führt das Skript unten zwei Kommandos im Hintergrund, einen zum Kopieren einer Datei (cp) und einen zum Kompilieren eines C-Programms (gcc) und einen Befehl im Vordergrund zum Bearbeiten einer Textdatei (vi) aus:

```
cp A B &
gcc x.c &
vi H
```

3.3.2. Shell-Programmierung

3.3.2.1. Die SH-Programmiersprache

In diesem Abschnitt werden wir die SH-Sprache vorstellen, die einfachste der UNIX-Shells. Wir werden die wichtigsten syntaktischen Kategorien betrachten. Die Semantik und Funktionalität jeder dieser Kategorien lassen sich aus dem Kontext leicht ableiten.

Wir werden uns auf die folgenden Regeln zur Beschreibung der Grammatik der SH-Sprache verlassen. Bitte beachten Sie, dass diese Regeln ausschließlich zur Angabe der Grammatik verwendet werden. Ähnliche Konstrukte werden später bei Dateispezifikationen und regulären Ausdrücke erscheinen, allerdings haben diese eine unterschiedliche Bedeutung.
Eine grammatikalische Kategorie kann durch eine oder mehrere alternative Regeln definiert werden. Die Alternativen werden wie folgt auf jeder Zeile geschrieben, beginnend auf der Zeile neben dem Namen der Kategorie:

GrammatikKategorie:
    Alternative 1
    ...
    Alternative N

[ ]? Bedeutet, dass der Ausdruck zwischen den Klammern höchstens einmal erscheinen kann
[ ]+ Bedeutet, dass der Ausdruck zwischen den Klammern mindestens einmal erscheinen kann
[ ]* Bedeutet, dass der Ausdruck zwischen den Klammern null oder mehrmals erscheinen kann

Das Bild unten zeigt die SH-Sprachen-Syntax auf einer höheren Ebene (ohne Eingabe viele Details) unter Berücksichtigung der oben genannten Ausdrücke. Die Bedeutung der syntaktischen Elemente im unteren Bild ist:
Wort: Sequenz von Zeichen ohne Leerzeichen (Leerzeichen oder Tab)
Name: Zeichenfolge, die mit einem Buchstaben beginnt und mit Buchstaben, Ziffern oder _ (Unterstrich) fortfährt
Ziffer: die zehn Dezimalstellen (0, ..., 9)

```
Befehl:
Grundbefehl
( Befehlsliste )
{ Befehlsliste }
if Befehlsliste then Befehlsliste [ elif Befehlsliste then Befehlsliste ]* [ else Befehlsliste ]? fi
case Wort  in [ Wort  [ | Wort  ]* ) Befehlsliste ;; ]+ esac
for Name do Befehlsliste done
```

```
for Name in [ Wort ]+ do Befehlsliste done
while Befehlsliste do Befehlsliste done
until Befehlsliste do Befehlsliste done

Grundbefehl:
        [ Element ]+

Befehlsliste:
        PipeKette [ Trenner  PipeKette ]* [ Ende ]?

PipeKette:
        Befehl [ | Befehl ]*

Element:
        Wort
       Name=Wort
        > Wort
        < Wort
        >> Wort
        << Wort
        >&Ziffer
        <&Ziffer
        <&-
        >&-

Trenner:
        &&
        ||
        Ende

Ende:
        ;
        &
```

Ein SH-Befehl kann jeden der oben beschriebenen neun Formen haben. Eine der Möglichkeiten, ihn zu definieren, ist Grundbefehl, wobei ein Grundbefehl eine Sequenz von Elementen ist und das Element in einer von zehn möglichen Möglichkeiten definiert ist. Ein PipedChain ist entweder ein Befehl oder eine Folge von Befehlen, die durch das Sonderzeichen '|' verbunden sind. cmdList ist eine Folge von PipedChains getrennt und gegebenenfalls mit Sonderzeichen beendet.

Die oben beschriebene Grammatik macht deutlich, dass SH Konstrukte ohne Semantik akzeptiert. Zum Beispiel kann der Befehl ein Grundbefehl sein, der ein einzelnes Element aus >&-; enthalten kann. Diese Eingabe wird von SH aus syntaktischer Sicht akzeptiert, obwohl sie keine Semantik besitzt.

SH-Shell hat die folgenden dreizehn reservierten Wörter:
if then else elif fi

case in esac
for while until do done

Die alternativen Strukturen von if und case werden durch fi beziehungsweise esac geschlossen, welche die Spiegelung ihres selbst sind. Bei sich wiederholenden Zyklen das Ende durch die Verwendung von reservierten done angezeigt. Nicht verwendet eine ähnliche Konstruktion entsprechend um zu tun, weil od nennt sich eine klassische Unix-Befehle.

Wir schließen diesen Abschnitt mit der Darstellung der Syntax einiger reservierter Konstrukte sowie ein paar Zeichen mit besonderer Bedeutung in der SH-Shell ab.

Wir schließen diesen Abschnitt mit der Syntax einiger reservierter Konstrukte, sowie eine Wenige Zeichen mit besonderer Bedeutung in SH.
Syntaktische Konstrukte:

|	Verbinden durch pipe
&&	logische UND-Verbindung
||	logische ODER-Verbindung
;	Separator / Befehl endet
;;	Falltrennzeichen
(), {}	Befehlsgruppierung
< <<	Eingabeumleitung
> >>	Ausgangsumleitung
&Ziffer, &-	Standard Eingangs- oder Ausgangsspezifikation

Vorlagen und generische Namen:

*	Jede Folge von Zeichen
?	Irgendein Zeichen
[...]	mit jedem Charakter in ...

Hinweis: Diese Vorlagen dürfen nicht mit den Grammatikstandards verwechselt werden, die am Anfang dieses Abschnitts vorgestellt wurden.

3.4. Aufgabenvorschläge

I.
Beschreiben Sie kurz die Funktionsweise des fork-Systemaufrufs und die Werte, die es zurückgeben kann.
Was wird unter Berücksichtigung, dass der fork-Systemaufruf erfolgreich war, auf dem Bildschirm zu sehen sein, wenn der Code unten ausgeführt wird? Begründe deine Antwort.

```
int main() {
        int n = 1;
        if(fork() == 0) {
                n = n + 1;
                exit(0);
        }
        n = n + 2;
        printf("%d: %d\n", getpid(), n);
```

```
        wait(0);
        return 0;
}
```

Was wird auf dem Bildschirm zu sehen sein, wenn das Shell-Skript-Fragment unten ausgeführt wird? Erklären Sie die Funktionsweise der ersten drei Zeilen des Fragments.

```
1  for F in *.txt; do
2      K=`grep abc $F`
3      if [ "$K" != "" ]; then
4          echo $F
5      fi
6  done
```

II.

Sehen Sie sich das Codefragment unten an. Welche Zeilen werden auf dem Bildschirm zu sehen sein und in welcher Reihenfolge, wenn man bedenkt, dass der fork-Systemaufruf erfolgreich ist? Begründe deine Antwort.

```
int main() {
        int i;
        for(i=0; i<2; i++) {
                printf("%d: %d\n", getpid(), i);
                if(fork() == 0) {
                        printf("%d: %d\n", getpid(), i);
                        exit(0);
                }
        }
        for(i=0; i<2; i++) {
                wait(0);
        }
        return 0;
}
```

Erklären Sie die Funktionsweise des Shell-Skriptfragments unten. Was passiert, wenn die Datei report.txt fehlt? Fügen Sie die Codezeile für die Erstellung der Datei report.txt hinzu.

```
more report.txt
rm report.txt
for f in *.sh; do
        if [ ! -x $f ]; then
                chmod 700 $f
        fi
done
mail -s "Betroffene Dateien melden" admin@scs.ubbcluj.ro < report.txt
```

4. Allgemeine Bibliographie

1.***: Linux man magyarul, http://people.inf.elte.hu/csa/MAN/HTML/index.htm
2.A.S. Tanenbaum, A.S. Woodhull, Operációs rendszerek, 2007, Panem Kiadó.
3.Alexandrescu, Programarea modernă in C++. Programare generică si modele de proiectare aplicate, Editura Teora, 2002.
4.Angster Erzsébet: Objektumorientált tervezés és programozás Java, 4KÖR Bt, 2003.
5.Bartók Nagy János, Laufer Judit, UNIX felhasználói ismeretek, Openinfo
6.Bjarne Stroustrup: A C++ programozási nyelv, Kiskapu kiadó, Budapest, 2001.
7.Bjarne Stroustrup: The C++ Programming Language Special Edition, AT&T, 2000.
8.Boian F.M. Frentiu M., Lazăr I. Tambulea L. Informatica de bază. Presa Universitară Clujeana, Cluj, 2005
9.Boian F.M., Ferdean C.M., Boian R.F., Dragoş R.C., Programare concurentă pe platforme Unix, Windows, Java, Ed. Albastră, Cluj-Napoca, 2002
10.Boian F.M., Vancea A., Bufnea D., Boian R.,F., Cobârzan C., Sterca A., Cojocar D., Sisteme de operare, RISOPRINT, 2006
11.Bradley L. Jones: C# mesteri szinten 21 nap alatt, Kiskapu kiadó, Budapest, 2004.
12.Bradley L. Jones: SAMS Teach Yourself the C# Language in 21 Days, Pearson Education,2004.
13.Cormen, T., Leiserson, C., Rivest, R., Introducere în algoritmi, Editura Computer Libris Agora, Cluj, 2000
14.Date C.J., Au Introduction to database Systems, Addison-Wesley, 1995
15.Eckel B., Thinking in C++, vol I-II, http://www.mindview.net
16.Ellis M.A., Stroustrup B., The annotated C++ Reference Manual, Addison-Wesley, 1995
17.Frentiu M., Lazăr I. Bazele programării. Partea I-a: Proiectarea algoritmilor
18.Herbert Schildt: Java. The Complete Reference, Eighth Edition, McGraw-Hill, 2011.
19.Horowitz, E., Fundamentals of Data Structures in C++, Computer Science Press, 1995
20.J. D. Ullman, J. Widom: Adatbázisrendszerek - Alapvetés, Panem kiado, 2008.
Kiadó Kft, 1998, http://www.szabilinux.hu/ufi/main.htm
21.Niculescu,V., Czibula, G., Structuri fundamentale de date şi algoritmi. O perspectivă orientată obiect., Ed. Casa Cărţii de Stiinţă, Cluj-Napoca, 2011
22.Raffai Mária: UML 2 modellező nyelvi szabvány, Palatia nyomda és kiadó, 2005.
23.Ramakrishnan R., Database Management Systems, WCB McGraw-Hill, 1998
24.Robert A. Maksimchuk, Eric J. Naiburg: UML földi halandóknak, Kiskapu kiadó, 2006.
25.Robert A. Maksimchuk, Eric J. Naiburg: UML for Mere Mortals, Pearson Education, 2005.
26.Robert Sedgewick: Algorithms, Addison-Wesley, 1984
27.Simon Károly: Kenyerünk Java. A Java programozás alapjai, Presa Universitară Clujeană, 2010.
28.Tâmbulea L., Baze de date, Facultatea de matematică şi Informatică, Centrul de Formare Continuă şi Invăţământ la Distanţă, Cluj-Napoca, 2003
29.V. Varga: Adatbázisrendszerek (A relációs modelltől az XML adatokig), Editura Presa Universitară Clujeană, 2005, p. 260. ISBN 973-610-372-2